2005-12

Implementation of a Configurable Fault Tolerant Processor (CFTP) using Internal Triple Modular Redundancy (TMR)

Majewicz, Peter J.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/1743

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A CONFIGURABLE FAULT TOLERANT PROCESSOR (CFTP) USING INTERNAL TRIPLE MODULAR REDUNDANCY (TMR)

by

Peter J. Majewicz

December 2005

Thesis Co-advisors:	Herschel H Loomis, Jr.
	Alan A. Ross

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Implementation of a Fault Tolerant Processor(CFTP) using Internal Triple Modular Redundancy (TMR)	**5. FUNDING NUMBERS**
6. AUTHOR(S) **Peter J. Majewicz**	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The environment of space is challenging to digital equipment due to the interaction between electrical systems and the radiation of space. One such effect is the Single Event Upset (SEU), which occurs when radiation causes a logical bit value to change. These effects are magnified in reconfigurable digital systems that utilize Field Programmable Gate Arrays (FPGA) because both the configuration and the data are susceptible to SEUs.

Several techniques have been developed in order to mitigate these effects. One such technique, called Triple Modular Redundancy (TMR), is an architecture where three identical systems perform the same operation in parallel. The three outputs are applied to a voter circuit which would eliminate an SEU caused error.

This thesis develops a five-stage pipelined Reduced Instruction Set Computer (RISC) microprocessor. A TMR architecture is then instantiated on an FPGA based circuit board. Instead of voting the processor outputs, the voting function is distributed and votes the outputs of all the internal pipeline registers. Even in the event of an SEU caused error, correct data is applied to the next pipeline stage. Finally this thesis describes and analyzes test data from radiation testing of the TMR system.

14. SUBJECT TERMS Field Programmable Gate Array (FPGA), FPGA testing, FPGA radiation testing, Single Event Upset (SEU), Triple Modular Redundancy (TMR)	**15. NUMBER OF PAGES** 109
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

IMPLEMENTATION OF A CONFIGURABLE FAULT TOLERANT PROCESSOR (CFTP) USING INTERNAL TRIPLE MODULAR REDUNDANCY (TMR)

Peter J. Majewicz
Lieutenant, United States Naval Reserve
B.S., Old Dominion University, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2005

Author:	Peter J. Majewicz
Approved by:	Herschel H. Loomis, Jr. Thesis Co-advisor
	Alan A. Ross Thesis Co-advisor
	Jeffrey B. Knorr Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The environment of space is challenging to digital equipment due to the interaction between electrical systems and the radiation of space. One such effect is the Single Event Upset (SEU), which occurs when radiation causes a logical bit value to change. These effects are magnified in reconfigurable digital systems that utilize Field Programmable Gate Arrays (FPGA) because both the configuration and the data are susceptible to SEUs.

Several techniques have been developed in order to mitigate these effects. One such technique, called Triple Modular Redundancy (TMR), is an architecture where three identical systems perform the same operation in parallel. The three outputs are applied to a voter circuit which would eliminate an SEU caused error.

This thesis develops a five-stage pipelined Reduced Instruction Set Computer (RISC) microprocessor. A TMR architecture is then instantiated on an FPGA based circuit board. Instead of voting the processor outputs, the voting function is distributed and votes the outputs of all the internal pipeline registers. Even in the event of an SEU caused error, correct data is applied to the next pipeline stage. Finally this thesis describes and analyzes test data from radiation testing of the TMR system.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would first like to thank my wife, Tina, and the children ,Amanda, Joey and Peter for their never-ending support and providing a hiatus from the rigors of school.

Additionally, special thanks are owed to the following individuals for their guidance and assistance.

To Professors Loomis and Ross for their continuing support and inspiration.

To Mindy Surratt whose tireless work brought the CFTP designs to life in hardware.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The environment of space is challenging to digital equipment due to the interaction between electrical systems and the radiation of space. The effect is manifested by faulty operation of the equipment. If the circuit is a component of a vital system (i.e., life support or thruster control) the result can be catastrophic. Many specific effects have been identified and are grouped into two main categories, Total Dose Effects and Single Event Effects (SEE) [1]. Total dose effects result from the accumulated exposure to radiation and typically produce the permanent degradation and eventual failure of a component. These effects are typically prevented by the “radiation hardening” of the component or by shielding it. Single Event Effects are more transient in nature and result in a temporary change in an electrical signal. A subset of SEE’s is Single Event Upsets (SEU), which occur when the incident radiation causes a logical bit value in a digital circuit to change. The goal of the Configurable Fault Tolerant Processor (CFTP) Project is to explore, develop and demonstrate the applicability of using off-the-shelf (COTS) Field Programmable Gate Arrays (FPGA) in order to design digital processor based systems that are resilient to SEUs.

Earlier research in the CFTP Program [1,2,3,4], developed a system which used three soft-core instantiations of a 16 bit Reduced Instruction Set Computer (RISC) processor and organized them in a technique known as Triple Modular Redundancy (TMR). The outputs of the processors were compared using a majority voter, so that if a fault occurs in one of the processors, only its output was affected and subsequently “voted out” when compared to the correct outputs of the other two processors. The correct signal is then applied to the destination (i.e. memory, output device). In order to correct the actual bit affected by the SEU, the detection of an error triggered the execution of an Interrupt Service Routine (ISR). The ISR first saved the contents of the internal register block, called the register file, to memory utilizing the voter circuit, and then wrote the corrected values back into the register file. At the conclusion of the ISR the contents of the register file of each of the three processors was assumed to be identical (free of errors). The originally running program resumed from a location based

on the address where the error was originally discovered. In addition to detecting a corrupted value, the voter also developed a coded output, named an *Error Syndrome* which gave information regarding which output signal and which one of the three processors contained the data affected by the SEU [4].

The original goal of this thesis was to continue the work previously mentioned. The first task was to expand the processor capability to handle 32-bit data words and 32-bit addresses. The goal was to implement a design similar to the format used in the five-stage pipelined MIPS processor developed in [5]. To increase the throughput of the processor, the design was to incorporate *data forwarding* and *hazard avoidance*. After processor operation was verified, a voter circuit was to be developed. Finally, the TMR architecture was to be assembled using three instantiations of the processor and the voter circuit.

This thesis describes the details of the *Pix* processor developed for implementation on the CFTP board. Additionally, the thesis describes a different voting mechanism than was originally proposed by previous thesis work. By analyzing the internal architecture of the processor, a system where the intermediate signals from each of the stage registers are voted was implemented. The result is that if an SEU occurs in one of these registers, a corrected output is automatically applied to the next stage register. The benefit of this method is that data errors are corrected *on-the-fly*. The running program can continue its execution without interruption. A secondary benefit gained from the voter design is that since voters are spread through-out the internal structure of the processor, more detailed information regarding the SEU is obtained. In addition to the above indications from the Error Syndrome, information detailing the phase of execution is also included.

This thesis also presents the issues involved with the actual programming and testing of the Pix TMR system on the CFTP board FPGA. This includes an interface circuit programmed into the second FPGA on the board which programs, initializes and receives input from the Pix TMR system and relays that data (either Memory data or Error Syndrome) to an output channel for monitoring. Testing of the Pix-TMR system is composed of simulations, a configuration error injection software tool and actual

radiation testing performed at the Crocker Nuclear Laboratory, University of California at Davis. The thesis concludes with a detailed analysis of the radiation test and recommendations for future work. Topics discussed in the test data analysis include the affect of an SEU on a configuration bit vice data bit, correlation between an upset of a configuration bit and its manifestation as data errors, and determining the status of the processor based on the *Error Syndrome* (if and when a reconfiguration is necessary).

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. CFTP OBJECTIVE

The objective of the CFTP Program at Naval Postgraduate School is to explore, develop and demonstrate the applicability of using commercial-of-the-shelf (COTS) Field Programmable Gate Arrays (FPGA) in the design of reconfigurable and reliable space-based digital systems. The ability to reconfigure a design and the assurance of reliable operation are paramount in the development of computer systems for space applications because they grant unprecedented flexibility to the designing engineer. Utilizing reconfigurable hardware allows the engineer to update a programmed circuit not only during the space craft design process, but potentially through out the orbital life of the system. As mission needs changes, the circuit configuration can be changed to maximize effectiveness and efficiency. Traditionally, devices specifically fabricated to perform reliably in a radiation environment have been used in system designs to ensure proper operation. The penalty for using these devices was that as compared to the state-of-the-art COTS alternative, cost and performance in terms of speed were sacrificed [2].

B. PREVIOUS WORK

Over a span of several years numerous students have contributed to the development of the CFTP program. Their work involved all aspects necessary for a successful program including research of the hazards in the space radiation environment, available technologies to develop an FPGA based printed circuit board, and finally research in the implementation of a reliable processor system. The following is a brief description of accomplished work.

1. Lashomb's Design

Peter A. Lashomb [1] described the basic components of a TMR system and their implementation on an FPGA. Additional fault mitigation techniques such as hardware redundancy, quadded logic and software redundancy were also discussed. The processor used in the TMR design was the soft core 8 bit micro-controller KCPSM available free-of-charge from Xilinx Corporation. The thesis included the design for the TMR structure, voter logic and error encoders.

2. Ebert’s Research

Dean A. Ebert’s work [2] consisted of a detailed explanation of the space radiation environment and their effects on electrical equipment. In pursuit of a reconfigurable System-on-a Chip (SOC) design, he provided a description of different available FPGA technologies, and all the required support components such as ROM and RAM memories. Figure 1 depicts the conceptual diagram of the hardware components and the TMR design used in the CFTP Program.

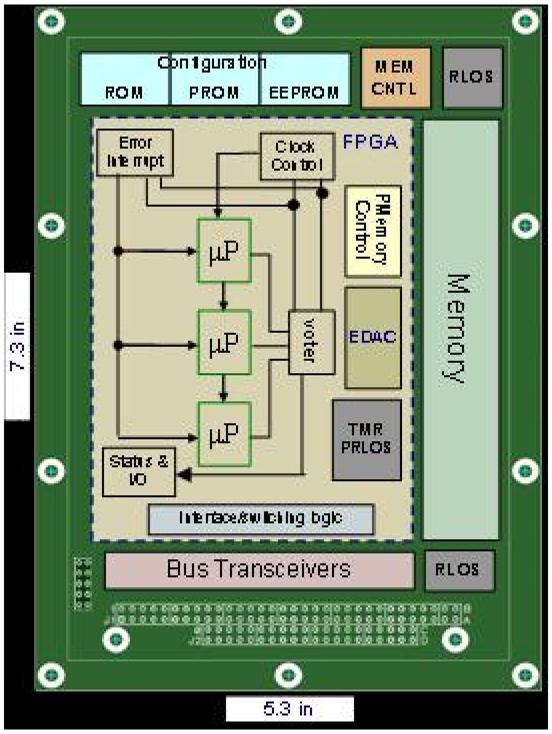


Figure 1. CFTP Conceptual Diagram (From Ref. [3])

Included in Ebert’s thesis is a complete description of the Department of Defense (DoD) Space Test Program (STP) and all the required actions for academic institutions like NPS to have the ability to launch their research experiments in space.

3. Johnson’s Implementation

Steven A. Johnson [3] concentrated his research in providing all the required components for a complete TMR system. The processor used in his design was a modified version of the 16 bit RICS DLX processor. His design included a state machine for implementing an Interrupt Service Routine (ISR) to recover from a fault, an *Error Syndrome Storage Device (ESSD)* to save the encoded Error Syndromes and a

Reconciler to provide a timing interface between the processor's Harvard architecture (separate buses for instruction and data memory) and off-chip memory's Von Neumann architecture (single bus). The system proposed by Johnson was based on theory. Simulation and timing issues were left as future work.

4. Yuan's System

Rong Yuan [5] implemented the system proposed by Johnson using Xilinx ISE design software and simulated it using ModelSim® simulation software. The bulk of the work was spent verifying the operation of the different components proposed in earlier work and resolving timing issues discovered in simulation. Considerable work was dedicated to the design and testing of the *Renconciler* which allowed the implementation of both Instruction and Data memories using a memory system with only one physical address and data bus, and the *Interrupt* mechanism which executed an ISR to reset the contents of the internal register file to correct values using the voter circuit and resetting the Program Counter to the value that it contained when the error was discovered.

C. HARDWARE DEVELOPMENT

1. CFTP Experiment Board

The result of the research was the production of the CFTP printed circuit board shown in Figure 2. A functional block diagram showing all connections is illustrated in Figure 3. The CTPF experiment board consists of 2 Virtex XQV-R600 SRAM FPGA's, a 500 KB EE-PROM memory which contains the configuration code for the first FPGA, a 4 MB Flash memory which contains the configuration of the second FPGA and 128 MB of SDRAM memory which can be used by the processor in the second FPGA. The first FPGA is directly connected to a PC-104 bus and acts as an interface to the space craft Command and Data Handler (C&DH). It is also directly connected to the other FPGA, which will hold the TMR processor system. The first FPGA's functions include configuring, initializing and exchanging data with the second FPGA. In operation, the first FPGA will monitor the operation of the circuit running on the second FPGA, exchange data with it and initiate a reconfiguration of the second FPGA when necessary. For these reasons, the first FPGA is referred to as the *Control* FPGA, and the FPGA containing the TMR processor is referred to as the *Experiment* FPGA [6].

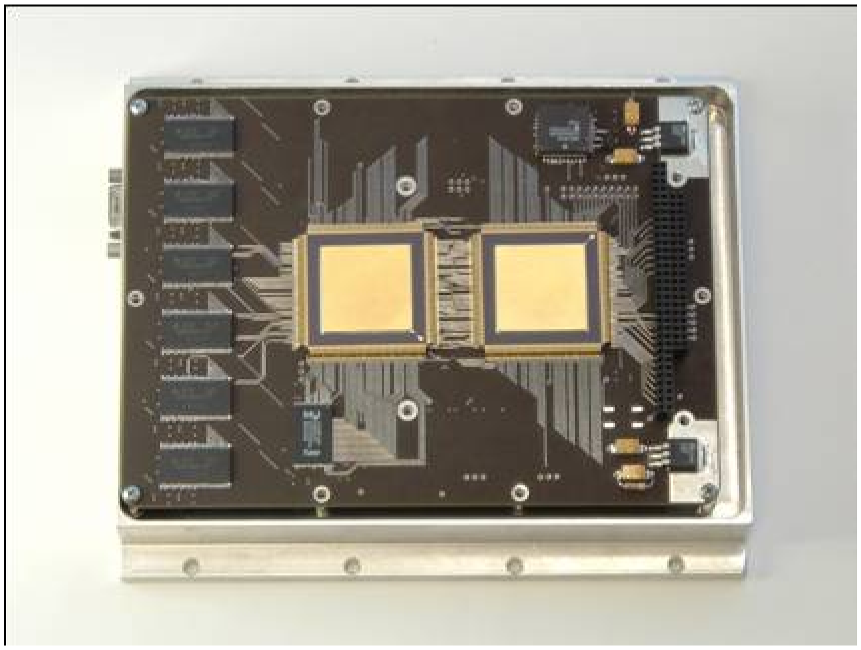


Figure 2. CFTP Printed Circuit Board

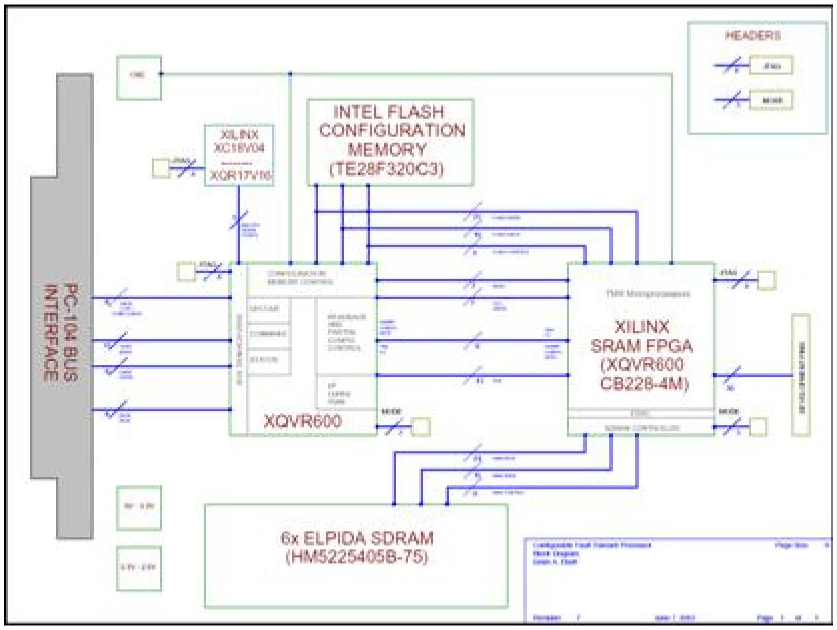

HEADERS
JTAG
MODE
INTEL FLASH CONFIGURATION MEMORY (TE28F320C3)
XILINX XC18V04
XQR17V16
PC-104 BUS INTERFACE
XQVR600
XILINX SRAM FPGA (XQVR600 CB228-4M)
6x ELPIDA SDRAM (HM5225405B-75)

Figure 3. CFTP PCB Functional Block Diagram / Schematic

2. CFTP II Experiment Board

Concurrent research at Naval Research Laboratory (NRL), Washington DC, led to the production of a very similar PCB, shown in Figure 4. The board has the same form-factor and PC-104 bus interface. The major difference is that the Experiment FPGA is a Virtex 2, and the board does not contain any SDRAM memory chips.

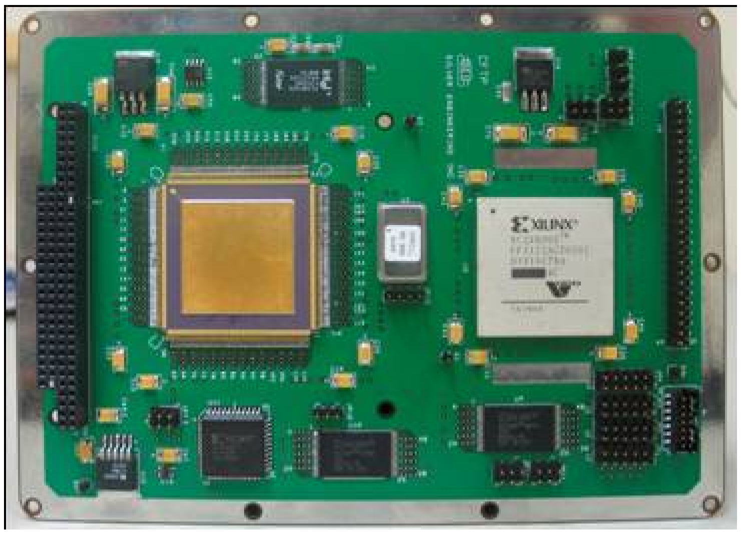


Figure 4. NRL Experiment PCB with Virtex 2 FPGA

D. CHAPTER SUMMARY

This chapter explained the main objective of the Configurable Fault Tolerant Processor (CFTP) Program, which is to design an FPGA printed circuit board to develop a fault tolerant processor system that will give engineers designing computer systems for space applications a high degree of confidence of reliable system operation, the ability to use COTS components in their circuit design and the ability to design their system knowing that future system configuration changes are possible.

The chapter also briefly described the work done by previous students in the CFTP Program. The work included research in the specific hazards and effects of the space environment, the various components required in the development of an FPGA based experiment PCB and finally a proposed TMR system.

The chapter ended with a description and gave illustrations of the actual PCB's produced by the CFTP Program and NRL for designing, implementing and testing the TMR processor system.

The next chapter describes the first step in implementing the proposed TMR system which is to expand the previously used KDLX processor to one that utilizes a 32-bit bus for internal processing and interfacing with memory (address and data). Instead of modifying the KDLX processor, the implementation commenced using an architecture developed by John Hennessy and David Patterson [5]. The resulting processor is referred to as *Pix* (Pete's Intelligent Experiment).

II. PIX PROCESSOR

A. DESCRIPTION

The Pix processor is a five stage, pipelined, RISC microprocessor with 32-bit data and address buses based on the MIPS microprocessor developed by John L. Hennessy and David A. Patterson [5]. The basic components of the processor are coded in VHSIC Hardware Description Language (VHDL). The components are then assembled together as modules in either VHDL or by using a graphical schematic entry tool. The software tool for the design and implementation of the Pix processor and its components is the Xilinx ISE Project Navigator, version 6.2.02i. The instruction set implemented resembles the instruction set of the MIPS 3000 architecture with the exception of instructions for addressing data less than 32 bits (i.e. Load Byte, and Load Half Word), supervisory mode instructions and floating point instructions. A complete listing of instructions, along with explanations and assembly code syntax is provided as Appendix A.

1. Instruction Decoding

The MIPS microprocessor was designed with a structured format for decoding instructions. To simplify the control and decoding circuits in the processor, all instructions are the same length, 32 bits. Additionally, the six most-significant bits (MSB), called the *Operation Code* (opcode) in every instruction indicate which one of the three main categories the instruction belongs. Table 1 shows a graphical breakdown of the three instruction types and the fields they contain. Table 2 lists all the Pix Instructions by type. The main difference between the instructions of each category is in the way that the remaining 26 bits are decoded.

Type:	Bits:					
	31 26	25 21	20 16	15 11	10 6	5 0
R	opcode = 0	Rs	Rt	Rd	shamt	funct
J	opcode	target				
I	Opcode	Rs	Rt	immediate		

Table 1. Instruction Formats

a. Register Instruction Format

If the opcode bits are all zero, the instruction is a *Register* type instruction (R Type). This instruction specifies three registers. Bits 25 through 21 and 20 through 16 specify the registers for the two source operands (Rs and Rt respectively), and bits 15 through 11 specify the destination register (Rd), where the result of the instruction is to be stored. The specific R type instruction is determined by decoding the least significant five bits of the instruction, known as the Function Field (funct). If the instruction is a shift, bits 10 through 6 indicate the magnitude of the shift else they are unused.

b. Jump Instruction Format

If the value in the opcode is equal to 2 or 3, the instruction is in the *Jump* format (J type). The lower 26 bits of this instruction specifies a target value used to compute a jump address (the 6 most significant bits are obtained from the Program Counter (PC).

It should be noted that the instruction set contains two Jump instructions which are not of the J Type. Jump Register (JR) and Jump and Link Register (JALR) both obtain their jump address from a register specified in the instruction. Therefore, they are of type R, with their op code value equal to zero.

c. Immediate Instruction Format

If the opcode of a valid instruction is not equal to 0, 2 or 3, the instruction is an *Immediate* type. This format specifies two registers. Bits 25 through 21 and 20 through 16 indicate the two operands used in the instruction. The 16 least significant bits specify an immediate constant used in the operation.

d. Not an Instruction

In the event an instruction is not recognized, the main control inserts all zeros as control signals. This is equivalent to a no operation (no-op or NOP) instruction. It has no affect on the processor as it passes through the pipeline. Currently there is no explicit indication to the user that an unrecognized instruction is encountered.

Type	Instruction	Assembly Code		Type	Instruction	Assembly Code
R Type	Add	ADD		**I Type**	Add Immediate	ADDI
	Add unsigned	ADDU			Add Immediate unsigned	ADDIU
	AND	AND			AND Immediate	ANDI
	Jump and Link, Register	JALR			Branch if equal	BEQ
	Jump, Register	JR			Branch if equal to zero	BGEZ
	NOR	NOR			Branch if equal to zero and Link	BGEZAL
	OR	OR			Branch if greater than zero	BGTZ
	Shift Left, Logical	SLL			Branch if less than or equal to zero	BLEZ
	Shift Left, Logical, Variable	SLLV			Branch if less than zero	BLTZ
	Set if Less Than	SLT			Branch if less than zero and Link	BLTZAL
	Set if Less Than, Unsigned	SLTU			Branch if not equal	BNE
	Shift Right, Arithmetic	SRA			Load word	LW
	Shift Right, Arithmetic, Variable	SRAV			OR Immediate	ORI
	Shift Right, Logical	SRL			Set if Less Than, Immediate	SLTI
	Shift Right, Logical, Variable	SRLV			Set if Less Than, Immediate, Unsigned	SLTIU
	Subtract	SUB			Store word	SW
	Subtract unsigned	SUBU			XOR Immediate	XORI
	XOR	XOR				
J Type	Jump	J				
	Jump and Link	JAL				

Table 2. Pix Instructions Listed by Type

2. Basic Components

Figure 5 illustrates the basic components of the MIPS processor. Operation begins with the Program Counter (PC), which contains the address of the instruction to be fetched from Instruction Memory. This portion of the complete instruction execution is referred to as the Instruction Fetch Phase (IF). The contents of the instruction are then decoded to determine the operation to be performed and the location of the operands required. This phase is termed the Instruction Decode Phase (ID). The values of the two source registers are loaded into the Arithmetic and Logic Unit (ALU). The Execution Phase (EX) is next, where the ALU executes the arithmetic or logical operation specified by the instruction on the operands applied to its inputs. The Memory Phase (MEM) is next in which a memory word is written to memory or retrieved from memory, depending on the instruction. Finally, the result of the ALU operation or a retrieved memory word is written to the specified destination register during the Write Back Phase (WB). It should be noted that not all instructions require the actions of each phase. For example, a *branch* instruction does not need to access memory or store a value back to a register.

The block labeled *Registers* in Figure 5 is an array of 32 registers and usually referred to as the *register file*. In the Pix processor, each register is 32 bits wide. All registers can be read from and written to except for Register 0, in which all 32 bits are tied to ground, making the data value of Register 0 always equal to zero, regardless if a data value is written to it. The data and control connections to the register file allow for reading from two registers and writing to a single register simultaneously.

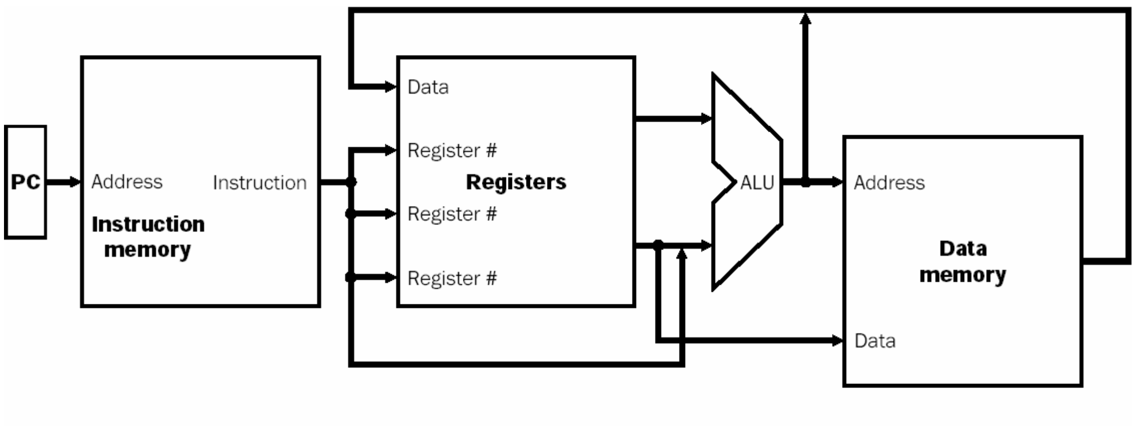


Figure 5. Basic Computer Components (From [2])

The amount of time (or the number of clock cycles) required to complete all of the phases required for an instruction varies with different architectures. A *Single-Cycle* architecture has all the phases completed in one clock cycle. A *Multi-Cycle* implementation typically involves a state machine and executes only one phase per clock cycle. Additionally, the state machine allots only the required number of clock cycles to complete an instruction. Finally, a *Pipelined* architecture allots the same number of clock cycles to every instruction (in the case of Pix 5 cycles/instruction) which allows the execution of sequential instructions to be overlapped. The Pipelined Architecture is used for implementing the Pix processor and will be explained in more detail in the following section.

3. Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution so as to increase the total number of instructions executed per unit time [5]. A common example used to describe the concept of pipelining is the chore of washing several loads of dirty laundry. The task of washing a load of laundry can be divided into five steps: (1) bring dirty clothes to the washing machine in a basket, (2) wash clothes in washing machine, (3) dry clothes in dryer, (4) fold clothes and finally (5) put clothes away. Suppose that there are enough dirty clothes for five loads and each step takes 30 minutes. If load one is put through all the steps before the next load is started, one load of laundry will take two hours and thirty minutes, and the entire five load chore will take twelve hours and thirty minutes. But if when the first load is done with step one and then the second load start the process, and all loads continue in the same manner, the time to wash the first load will still take two hours and 30 minutes, but the entire five load chore will take only four hours and thirty minutes to complete. In general, assuming that there is always an item to put into the first step, the speedup due to pipelining is equal to the number of stages in the pipeline [5]. Another way to visualize the efficient operation of a pipelined processor is that after the first instruction completes its operation on the fifth clock cycle, all ensuing instructions complete their operations every subsequent clock cycle because of the overlapping execution.

The first step to implement a pipelined architecture is to place a register at the transition boundary between phases. Figure 6 illustrates these locations. The primary purpose of these registers is to maintain data that is required in the future phases. For example, the destination register for an ADD instruction is decoded in the IF phase. The data is not written to the destination register until the WB phase, three clock cycles later. The five bits that represent the destination register must be stored in each phase register until it is used in the WB phase.

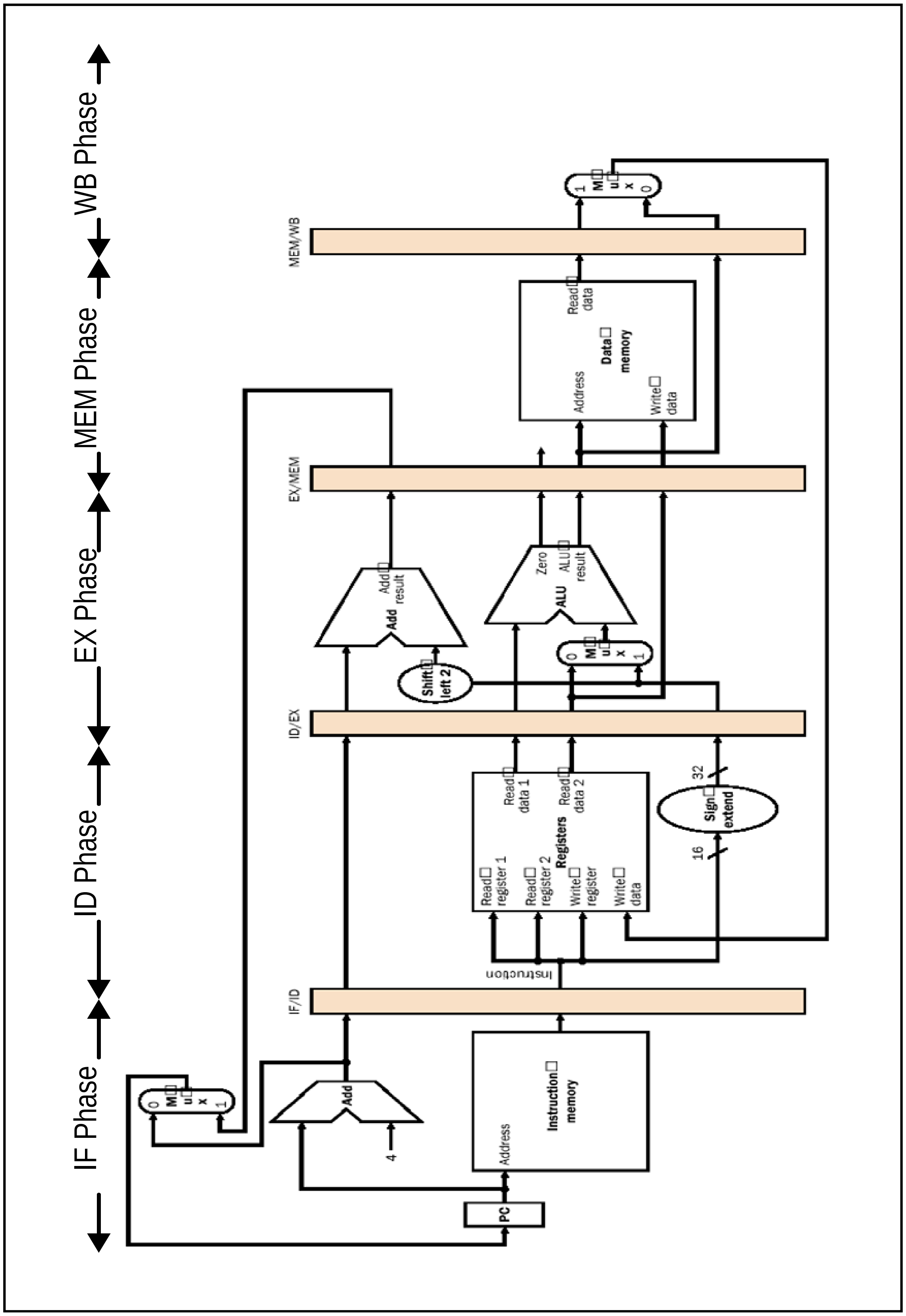


Figure 6. Processor with Pipeline Registers (After [5])

An additional purpose for the phase registers is to store *control* data. As the instruction is being decoded in the ID phase, a *Control* module determines what signals are needed at that time and during the subsequent phases of execution. Examples of control signals include: ALU control signals which determine what operation is going to be performed, and select control signals for the multiplexers used throughout the processor (only two shown in Figure 6 for simplicity). Figure 7 illustrates graphically how the control signals are generated and then stored in successive registers until they are needed. The Pix *Control Unit* will be discussed in more detail later.

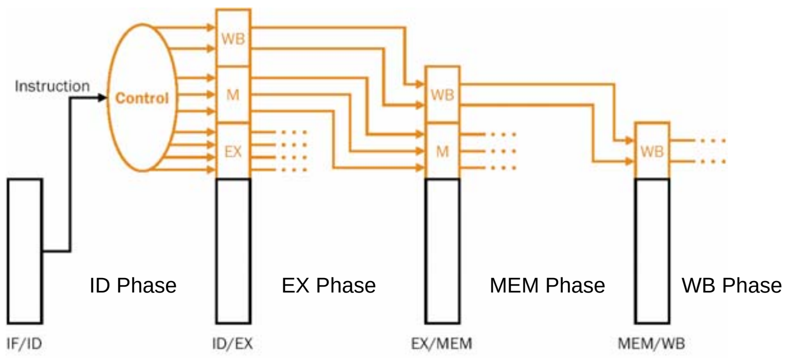


Figure 7. Control Lines for Final Three Phases (After [5])

4. Hazards

In pipeline architectures, there are two frames of reference that must be tracked when analyzing the internal operations. First is the number of clock cycles it takes to complete an instruction. In the Pix processor, all instructions are allotted five clock cycles for completion. The next frame of reference deals with the fact that during any single clock cycle, up to five instructions may be involved in different phases of execution. This can lead to problems for example, when the execution of one instruction relies on the completion of a previous instruction (which hasn't completed yet). These conditions are referred to as *hazards* and include *structural*, *control* and *data*.

a. Structural Hazards

Structural hazards occur when hardware cannot support the required actions of two instructions at the same time. Our use of the structured five stage architecture proposed in [5] eliminates the vast majority of these hazards. The only circumstance where structural hazards may be an issue is for memory access. The system must support two independent accesses to memory; one for the instruction, the other for data. In previous CFTP designs, this was accomplished using the *Reconciler*. In the Pix implementation, a dual port RAM memory is instantiated on the FPGA allowing two simultaneous accesses.

b. Control Hazards

Control hazards occur when the actions to be executed depend on the result of a not-yet completed action. An example is a *branch* if the value in R1 is greater than zero. The hazard arises from the fact that the determination of the value comes from the ALU and is available in the MEM phase. If the branch is taken (new value for the PC) then three improper instructions entered the pipeline. If the branch is not taken, then the next three sequential instructions are the correct ones to execute. To minimize the affect of having incorrect instructions starting execution, the Pix processor implements a *Comparator* circuit in the ID stage which immediately determines if a branch should be taken or not. If not taken, only one incorrect instruction enters the pipeline, and a *Control* signals clears it.

c. Data Hazards

Data hazards occur when an action requires a data value which is the result of a not-yet completed instruction. The following instruction sequence offers an example. The syntax is such that the three registers are listed as: [destination],[source],[source]. So line 1 adds register R2 to R1 and the sum is stored in register R3.

```
1.      ADD R3, R2, R1
2.      ADD R5, R4, R3
```

The addition in line 2 assumes that the execution of line 1 is complete (value is saved in R3). That is, the final value in R5 should be (R1 + R2) + R4. The problem is that the value of R2 + R1 is not saved in R3 until two clock cycles later, which is after the two values for the second instruction enter the ALU. This problem can be solved by either stalling the second instruction until the first one is finished, or by designing a forwarding network which forwards the result from the first instruction as soon as it is available and applies that value into the corresponding ALU input. The Pix processor uses a forwarding network which will be described in more detail. There is a situation in which forwarding does not work. If a value is required from memory, the instruction must wait until phase four (MEM) for that value to be used. In this case, a stall will be inserted into the pipeline.

5. Dealing with Hazards in the Pix Processor

Control and Data Hazards are handled by one of two modules, either the *Forwarding Unit* or the *Hazard Unit*.

a. Data Forwarding

The *fwd_unit* is the module in the ID phase that determines if a data hazard exists and the proper action to take. Data hazards generally apply only to the input values of the ALU, with the exception of a memory transfer, discussed shortly. Referencing the example addition code above, to obtain the correct value, the addition corresponding to line 2 can be stalled for two clock cycles, or the value of R2 + R1 can be forwarded to the input of the ALU in time for the addition corresponding to line 2.

The following VHDL code is used to implement the logic of the *fwd_unit.* The first section of code (lines 1-3) relates to a memory transfer situation where a data value is retrieved and saved to a register, then immediately saved back to memory, usually to a different address. Line 1 tests to see if the instruction in the WB phase is writing a value to a register that has recently been retrieved from memory, and if the instruction in the MEM phase is an instruction to save a value to memory. Line 2 tests to see if the two registers used are the same. If they are, then the output *Fwd_C* is asserted. This output is the *select* input to a multiplexer which *forwards* the newly retrieved value from the WB phase to the MEM phase to be written back into memory.

```
            -- SW after LW using same register (mem-to-mem Xfer)
1.          if    (MEM_WB_Mem2Reg = '1') AND (EX_MEM_MemWrite = '1')
2.                 AND (MEM_WB_Rd = EX_MEM_Rd) then Fwd_C <= '1';
3.          end if;

            -- Rs Register
            -- ID Hazard
4.           if   (ID_EX_RegWrite = '1') AND (ID_EX_Rd /= "00000")
5.                AND (ID_EX_Rd = IF_ID_Rs) then Fwd_A <= "01";
            -- EX Hazard
6.          elsif (EX_MEM_RegWrite = '1') AND (EX_MEM_Rd /= "00000")
7.                AND (EX_MEM_Rd = IF_ID_Rs) then Fwd_A <= "10";
            -- MEM Hazard
8.          elsif (MEM_WB_RegWrite = '1') AND (MEM_WB_Rd /= "00000")
9.                 AND (MEM_WB_Rd = IF_ID_Rs) then Fwd_A <= "11";
10.         end if;

            -- Rt register
            -- ID Hazard
11.         if   (ID_EX_RegWrite = '1') AND (ID_EX_Rd /= "00000")
```

```
12.             AND (ID_EX_Rd = IF_ID_Rt) then Fwd_B <= "01";
           -- EX Hazard
13.        elsif    (EX_MEM_RegWrite = '1') AND (EX_MEM_Rd /= "00000")
14.              AND ((EX_MEM_Rd = IF_ID_Rt) OR EX_MEM_Rd = ID_EX_Rt)
15.          then Fwd_B <= "10";
           -- MEM Hazard
16.        elsif (MEM_WB_RegWrite = '1') AND (MEM_WB_Rd /= "00000")
                 AND (MEM_WB_Rd = IF_ID_Rt) then Fwd_B <= "11";

17.        end if;
```

In the second section of code (lines 4-10), the instructions in the ID, EX, MEM are checked to see if they involve writing a value to a register. If they do, the register number is compared to see if it matched the register number Rs for the instruction currently in the Instruction Register (IR). If there is a match, the value that will be written to the register is forwarded to ALU through a multiplexer. The instruction in the WB phase is not checked because its value will be saved in time for the instruction in the IR to use. The order of checking phases is important in this situation. If a match exists with the value in the ID phase, its value is used since it will be the most up-to-date. The check to see if a register number is equal to zero is done because if any value is written to R0, the value in R0 remains equal to zero.

The third section of code (lines 11-17) consists of the same checks as the second section except they compare the values to the Rt register specified by the instruction in the IR.

b. Hazard Unit

The *hazard_unit* module determines if the processor needs to be stalled. The Pix processor was implemented so that only one situation necessitates a stall, that is if one instruction loads a data value from memory and the very next instruction uses that value for anything other than a load to memory. In this situation, the data value is retrieved from memory in the MEM phase of the first instruction. The value is needed in the ID phase of the next instruction. Since the value is not yet retrieved from memory, it cannot be forwarded to the ID phase. The second instruction must be *stalled*. This is accomplished by nullifying the second instruction (currently in the IR) by clearing the control bits in the IF/ID Register and reloading the same instruction into the IR during the following clock cycle. The original instructions in the EX, MEM, and WB continue unhampered, so the value needed by the instruction becomes available. The following section of VHDL code is from the *hazard_unit* module. The first three lines apply an external stall signal to the processor. Lines 4 and 5 test to see if the condition for a stall is met. Both the Rs and Rt registers in the IF/ID register need to be checked. If conditions warrant, the following outputs are asserted: *Stall_PC*, so the PC is not incremented, *Stall_IF_ID*, to disable the IR so that the current instruction (#2 from example) is not overwritten, and IF_Flush, to clear the control bits of the *stalled* instruction. The effect is that the second instruction is once again present in the ID phase, this time with an available data value. Stalling the instruction is also known as inserting a *bubble* into the pipeline [2].

```
1.        if Stall_in = '1' then             -- external input to STALL processor
2.        Stall_PC    <= '1';                -- disables loading of new PC value
3.        Stall_IF_ID <= '1';                -- disables loading of new instruction

          -- Load Instruction       -- Following Store Instruction not Hazard
4.        elsif (ID_EX_MemRead = '1') AND (IF_ID_MemWrite = '0')

          -- next instruction uses Reg being loaded
5.        AND ((ID_EX_Rt = IF_ID_Rs) OR (ID_EX_Rt = IF_ID_Rt))    then

6.        Stall_PC    <= '1';              -- disables loading of new PC value
7.        Stall_IF_ID <= '1';              -- disables loading of new instruction
8.        ID_Flush    <= '1';              -- control signals cleared -> NOP inserted
9.        end if;
```

B. PIX IMPLEMENTATION

The following sections will describe the implementation of the Pix processor by phase.

1. Instruction Fetch Phase

As discussed earlier, the IF phase is when the instruction pointed to by the PC is fetched from the Instruction Memory. Figure 8 is a schematic of the components which compose the IF phase.

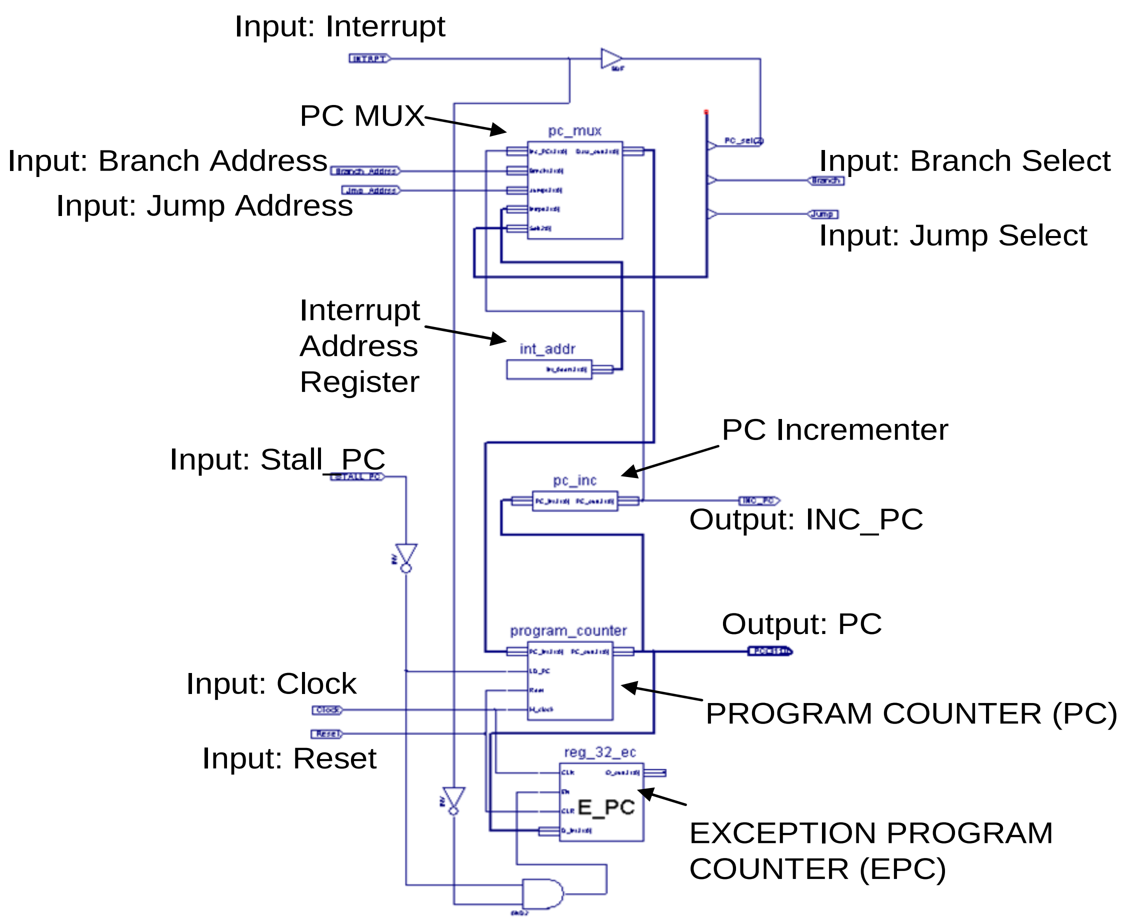


Figure 8. Instruction Fetch Phase of Pix Processor

The Program Counter is a 32-bit register which is used to hold the address of the instruction being fetched from memory. During normal operation, instructions are fetched sequentially by having the value in the PC incremented, pass through the PC MUX and loaded back into the PC at the next clock edge. The PC MUX module allows four different sources to become the value loaded in the PC; allowing for non-sequential instruction execution. A second source for the PC is a branch address. If the Branch Select input is asserted, the MUX will forward the branch address input to the PC. A similar action results if the Jump Select input is asserted, only a Jump address is forwarded to the PC. The last source for the PC is the address from the Interrupt Address Register which will make the PC point to the first instruction of an Interrupt Service Routine (ISR). The select logic for the PC MUX is designed so that if the Interrupt select input is asserted, it has priority over the other inputs. The Exception PC holds the value of the PC and is used to determine the address of an instruction which causes an Exception. Finally, the incremented PC value is an output of the IF stage and applied to the IF/ID register. This value is used to calculate branch addresses in the ID phase, so it must be stored for future use. As mentioned earlier, the PC output signal is applied to the Instruction Memory.

2. Instruction Decode (ID) Phase

The Instruction Decode phase is the portion of execution where the instruction in the IF/ID register is broken up into its individual fields and decoded. The following is a schematic of the Instruction Decode Phase.

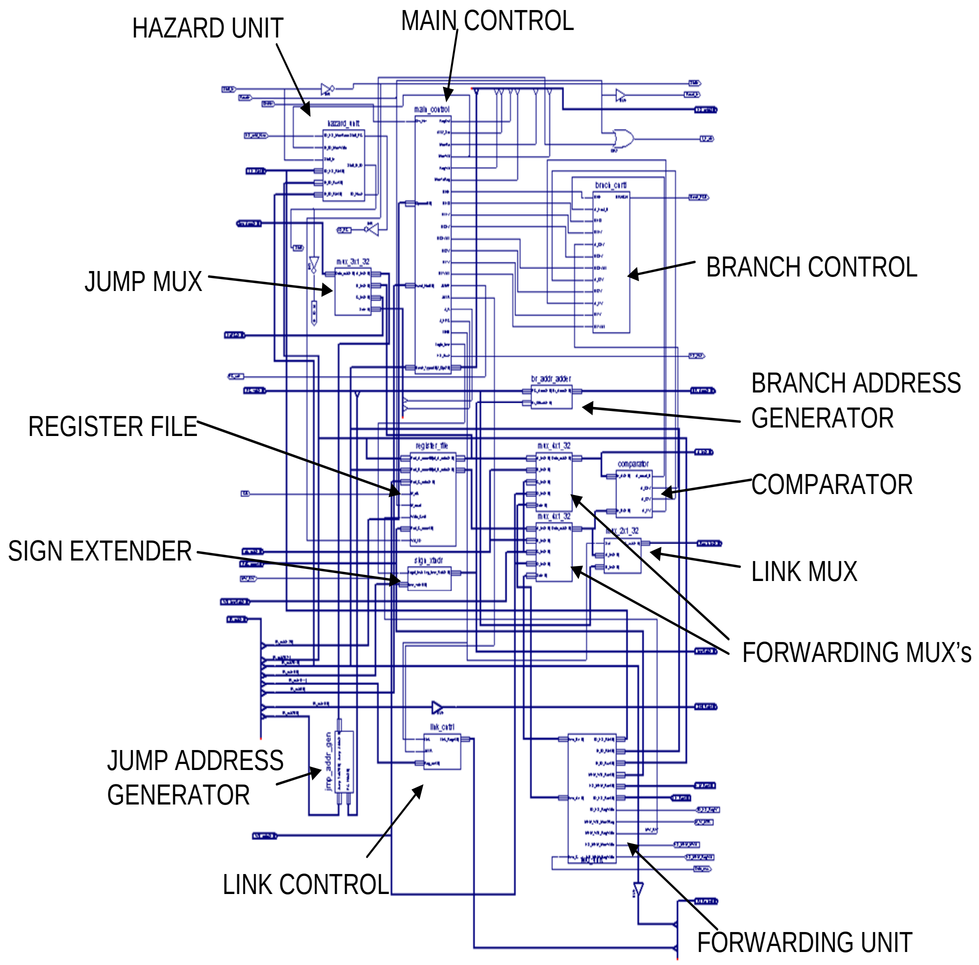


Figure 9. Instruction Decode Phase

The key element of the ID phase is the Main Control Unit (MCU). It interprets the fields of the current instruction to determine the current operation. As discussed earlier, it generates the appropriate control signals and forwards them to ID/MEM register. If the current instruction is a branch, it will specify the type of branch to the Branch Control Unit (BCU). The BCU will then take the signal from the Comparator which gives relative information on the two inputs to the ALU, and initiate a branch if necessary. For example, if the MCU asserts the Branch if Equal (BEQ) signal to the BCU and the Comparator asserts the A_equal_B signal, the BCU asserts the Branch_PC output. The result is that the branch address generated by the Branch Address Generator (BAG) is applied to the PC MUX back in the IF Phase. The Branch_PC output is applied as the select input to the PC MUX so that the branch address is loaded into the PC at the next clock edge. The branch address is computed by taking the lower sixteen bits of the current instruction (referred to as an offset value) sign extending them to 32 bits, then adding it the stored PC value. The MCU also determines if the instruction is a Jump. Since Jumps are unconditional, the Comparator is not used. If determined to be a Jump, the MCU will send a select signal to the PC MUX so that the Jump address is loaded into the PC. The JUMP MUX determines which of the three types of Jump addresses is sent to the PC MUX. For a standard Jump instruction, the address is the output of the Jump Address Generator (JAG), which concatenates the 26 lower bits of the instruction and the six most significant bits of the PC. If the instruction is a Jump Register, the Jump address is the value in the register specified in the instruction. If the instruction is a Return from Exception, the Jump address is the value in the Exception PC. Finally the MCU must determine if a Jump or Branch instruction is also a *Link* instruction. If it is, the PC value from the IF/ID Register is saved in a register in the Register File designated in the instruction. If no register is specified, Register 31 is used as a default. The Link Control module determines what the correct destination register is. The Link MUX forwards the PC value to the MEM stage so that it will eventually be written in the Register File during the WB phase.

The Forwarding Unit, Hazard Unit, and Register File reside in the IF phase and have been previously described.

3. Execution (EX) Phase

The Execution phase is where the arithmetic and logical operations take place. The result of the operation could be a value to be stored in a register or could be the effective address for a memory read or write operation. Figure 12 is a schematic showing the components of the EX phase.

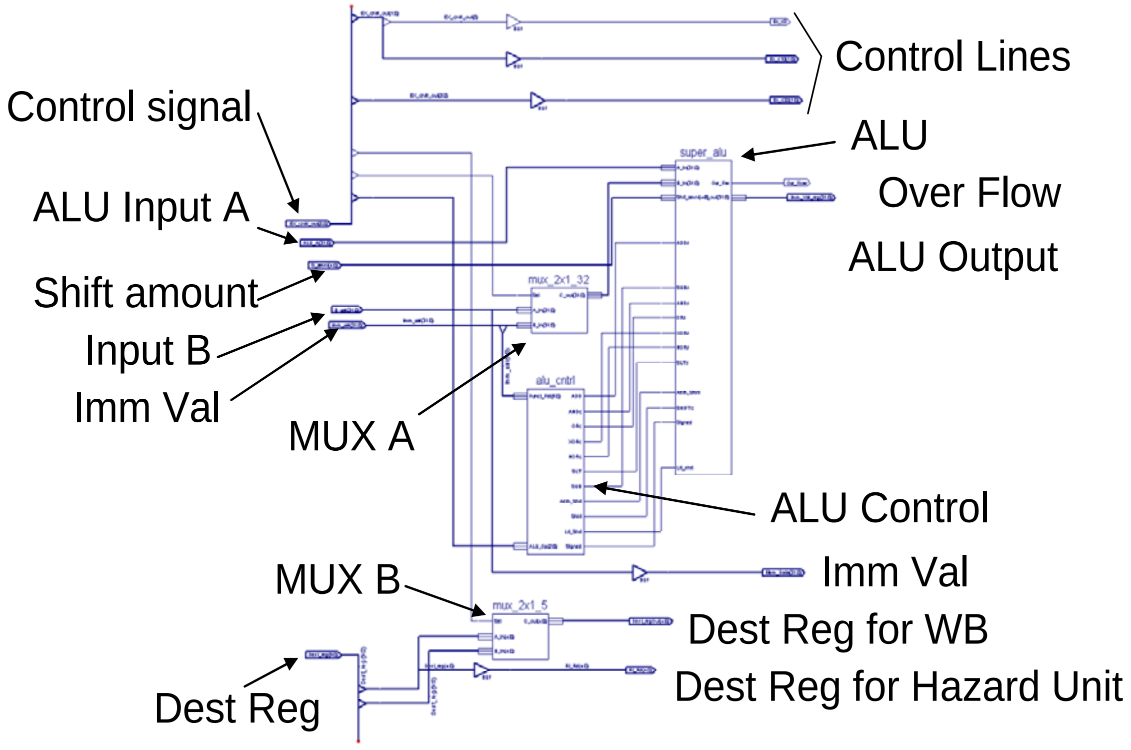


Figure 10. Execution Phase Schematic

The combinatorial logic that performs the mathematical and logical operation is contained in the Arithmetic and Logic Unit (ALU). The specific operation performed by the ALU is determined by the ALU Control module. It receives input from two sources. The first is a three bit control word generated by the Main Control module in the ID phase. ALU Control module first decodes this input to determine what operation the ALU is to perform. In case of an R type instruction, the ALU Control module must then decode the six bits of the Function Field (its other input) to determine the operation. The VHDL code for the ALU Control module is shown below. Lines 1 through 10 form the first case statement which decodes the three bit ALU Operation input and determines what the corresponding output should be. In the case when the input is equal to *010* (line 10), the ALU control module must then decode the six bits of the Function Field which is coded in the second case statement, lines 11 through 35.

```
case ALU_Op is
        when "000" => ADD  <= '1';              -- ADD for LW, SW, ADDI or ADDIU instr
        when "001" => SUB  <= '1';              -- SUB for Branch
        when "011" => ADD  <= '1';
                      Signed  <= '1';           -- allow Over Flow
        when "100" => ANDc <= '1';              -- ANDI instr
        when "101" => ORc   <= '1';             -- ORI instr
        when "110" => XORc <= '1';              -- XORI instr
        when "111" => SLT    <= '1';            -- SLTI or SLTIU instr
        when "010" =>                  -- R type instr =>
        case Funct_Fld is                           -- look at Function Field
                when  "100000" =>     ADD    <= '1';     -- 32
                                      Signed <= '1';         -- allow Over Flow
                when  "100001" =>     ADD    <= '1';     -- 33 ADDU
                when  "100010" =>     SUB    <= '1';     -- 34
                                      Signed <= '1';     -- allow Over Flow
                when  "100011" =>     SUB    <= '1';     -- 35 SUBU
                when  "100100" =>     ANDc  <= '1';      -- 36
                when  "100101" =>     ORc    <= '1';     -- 37
                when  "100110" =>     XORc  <= '1';      -- 38
                when  "100111" =>     NORc  <= '1';      -- 39
                when  "101011" =>     SLT     <= '1';    -- 42
                when  "101010" =>     SLT     <= '1';    -- 43 SLTU
                when  "000000" => Lft_Shft     <= '1';   -- 00
                                  Shift         <= '1';
                when  "000010" =>   Shift        <= '1';   -- 02
                when  "000011" =>  Arith_Shft  <= '1';   -- 03
                                     Shift      <= '1';
                when  "000100" => Lft_Shft      <= '1';  -- 04
                                     Shift       <= '1';
                when  "000110" =>   Shift         <= '1';  -- 06
                when  "000111" =>   Arith_Shft  <= '1';  -- 07
                                    Shift       <= '1';
                when others     => null;
                end case;
        when others    => null;
        end case;
```

MUX A determines what value is applied to the second input of the ALU. The possible value is either from the register specified in the instruction (R type instruction) or a 16 bit immediate value from the instruction (I type) that has been extended to 32 bits. For arithmetic values the extension preserves the sign of the 16 bit value, for a logical operation, the value is extended with zeroes.

MUX B applies the correct value for the destination register to be forwarded to the MEM and WB stages. For R type instructions, the destination register is determined by decoding bits 15 through 11 of the instruction. For I type instructions, the register value is in bits 20 through 16. The select signal is generated by the Main Control in the ID phase and stored in the ID/EX register.

The top three lines of the schematic represent five bits of control data which are not used in the EX phase, but forwarded to the MEM or ID phases. Of the five bits, one bit is applied to the ID phase to the fwd_unit, and four bits are stored in the EX/MEM register (two as control bits for the MEM phase and two for the WB phase).

The 32-bit output *Imm Val* is the value which will be written to memory if the instruction is a *store word* (SW). This value comes directly from the register file or from the forwarding network. The architecture does not allow for the output of the ALU to be written directly to memory. For a SW or load word (LW), the value out of the ALU is the effective address used by the memory instruction.

Detailed schematics and VHDL module code are provided in Appendix B.

4. Memory (MEM) and Write Back (WB) Phases

There is very little logic included in the MEM and WB phases. As the name implies, the MEM phase is where the processor interfaces with data memory. The two control signals consist of a *Write* and *Read* signal and control the operation being executed. One MUX is contained in the MEM phase that allows the immediate writing of a memory data word previously retrieved from memory, back to memory (usually to a different address). A second MUX is used to take one of the two sources destined to be stored during the WB phase and forward it to the *fwd_unit*. The WB phase is the final phase of execution for most instructions. Data that has been processed in the ALU or retrieved from memory is applied to the *register file* for storing in the destination register specified in the instruction.

5. Assembled Pix Components

Figures 11 and 12 are schematics of the complete Pix processor. Figure x shows the individual modules represented as blocks. Each block in the diagram was implemented in VHDL. The Xilinx ISE design allows the representation of a module as a block, which can then be inserted in another schematic. To form a simplified version of the Pix processor, all the components of each phase were then grouped together and made into a schematic block. Figure y is logically the same circuit, but with major components shown as blocks.

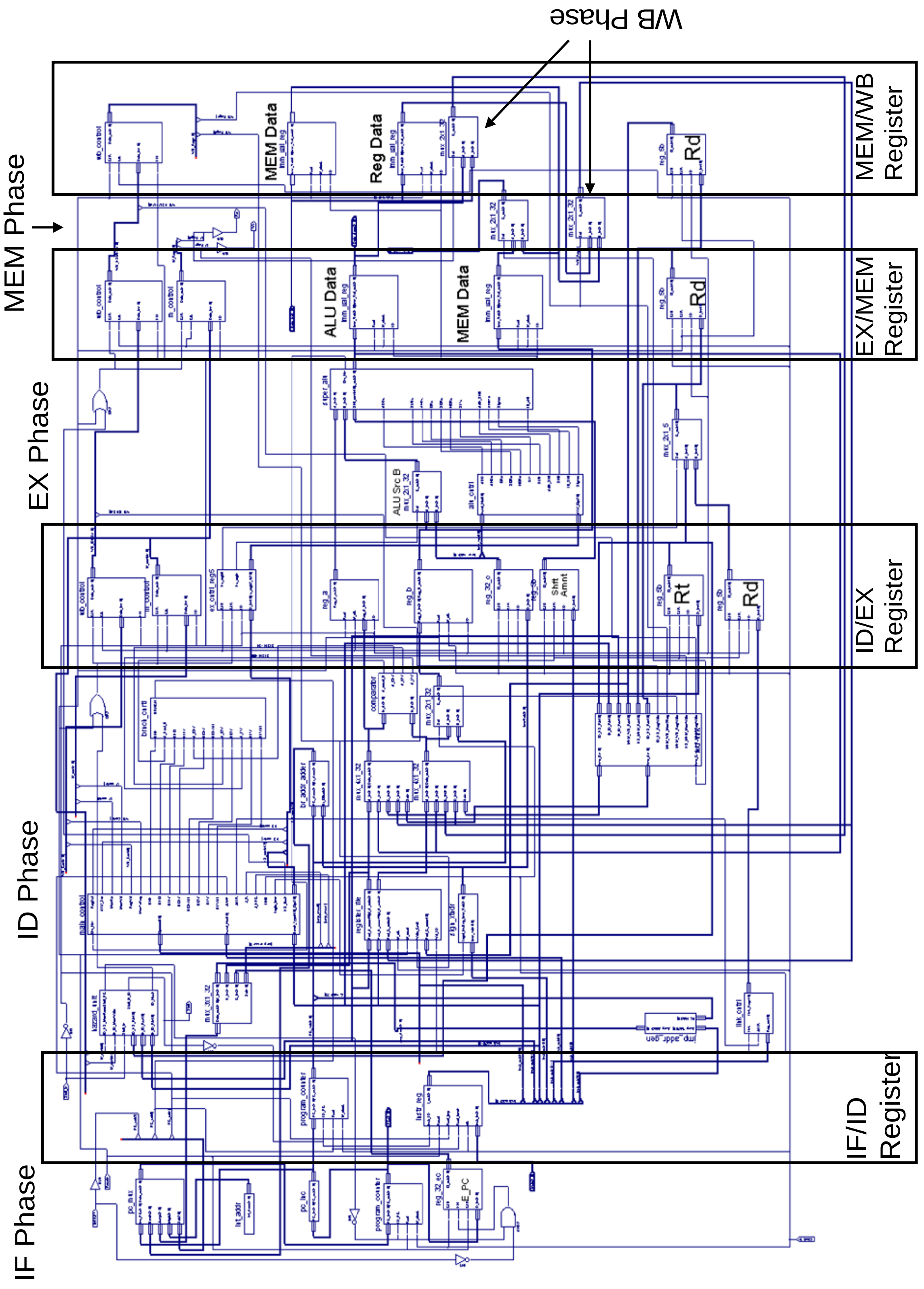


Figure 11. Schematic of Pix Processor

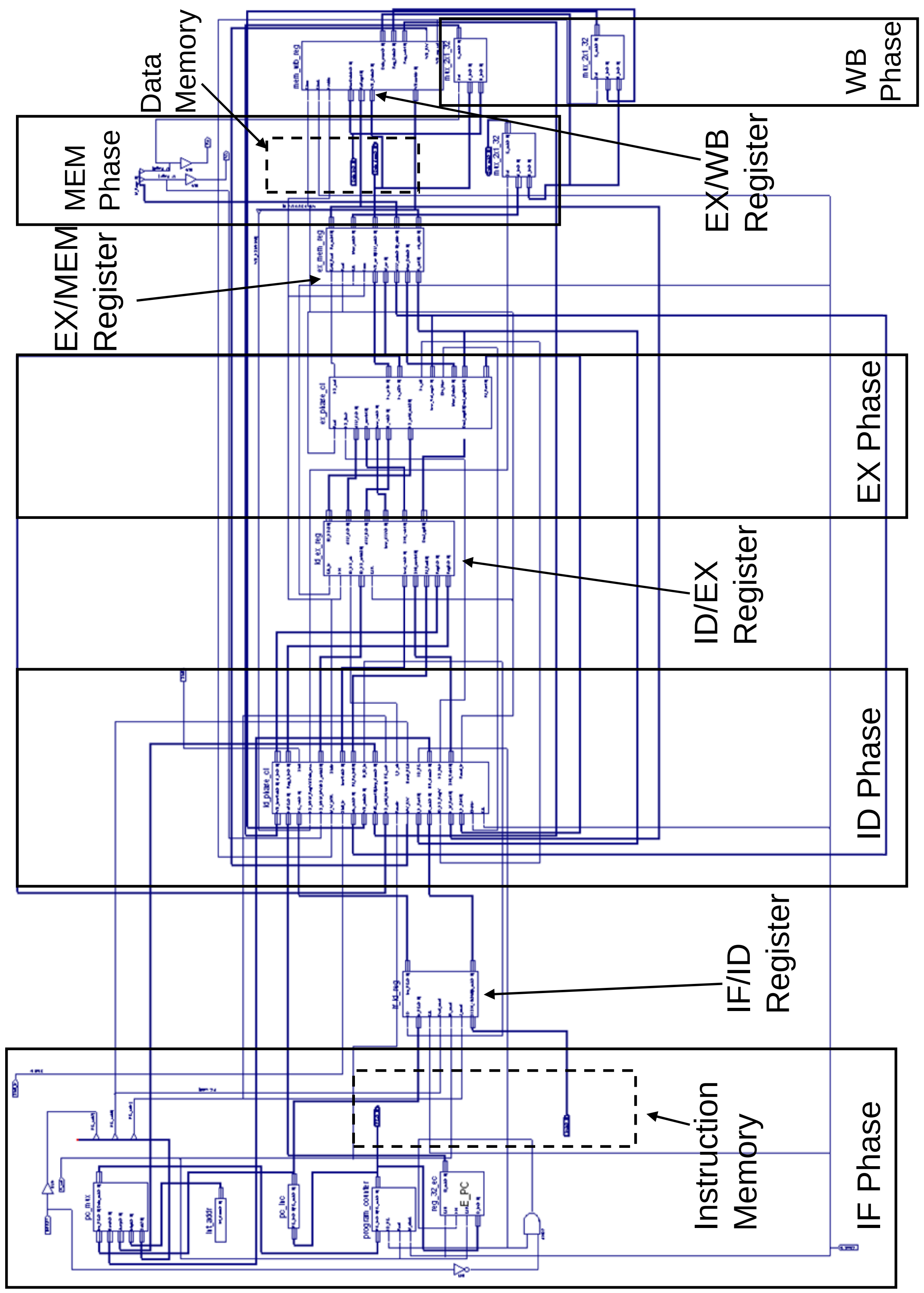


Figure 12. Schematic of Pix with Major Components as Blocks

C. CHAPTER SUMMARY

This chapter has explained the implementation of the five-phase pipelined Pix processor developed for used in the TMR system. Detailed descriptions of the fetch, decode, execution, memory and write back phases were included for both a generic processor and for the Pix processor. All schematics and VHDL codes pertaining to the Pix processor are contained in Appendix B.

The next chapter begins with an explanation of a simple voting circuit and quickly expands the design to describe the complete TMR system incorporating three copies of the Pix processor. The initial design developed in [4] is briefly mentioned to differentiate the design developed in this thesis.

III. TMR ARCHITECTURE

A. VOTING LOGIC

The key element in any TMR architecture is the voter. It is a circuit composed of combinatorial logic which implements a *majority* function. The circuit has three inputs, which are assumed to be the outputs of three identical circuits that provide the same value. During normal operation the outputs of these three identical circuits will match. If a fault occurs in one of them, its output may change. The voter compares the three inputs and produces an output which is equal to the input value in the majority. Figure 13 shows a schematic for a simple one-bit voter circuit. Table 3 is a truth table showing the output of the voter circuit for every combination of inputs.

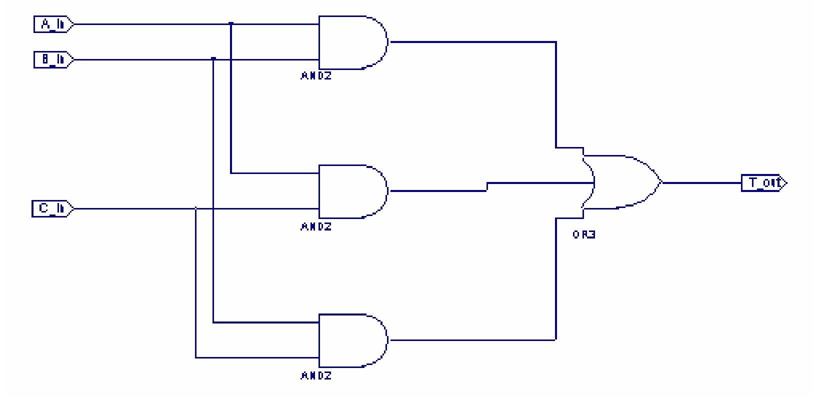


Figure 13. Single Bit Simple Voter

A	B	C	T_OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	1	1

Table 3. Truth Table for Single Bit Voter

The circuit shown in Figure 14 can be easily expanded in order to provide an indication of an error occurring. Three input AND and three-input NOR gates are used to determine if there is a mismatch among the inputs. If there is, the ERR output is asserted. This signal can then be used to initiate an *error routine* or similar action. A schematic for this circuit is shown in Figure 14.

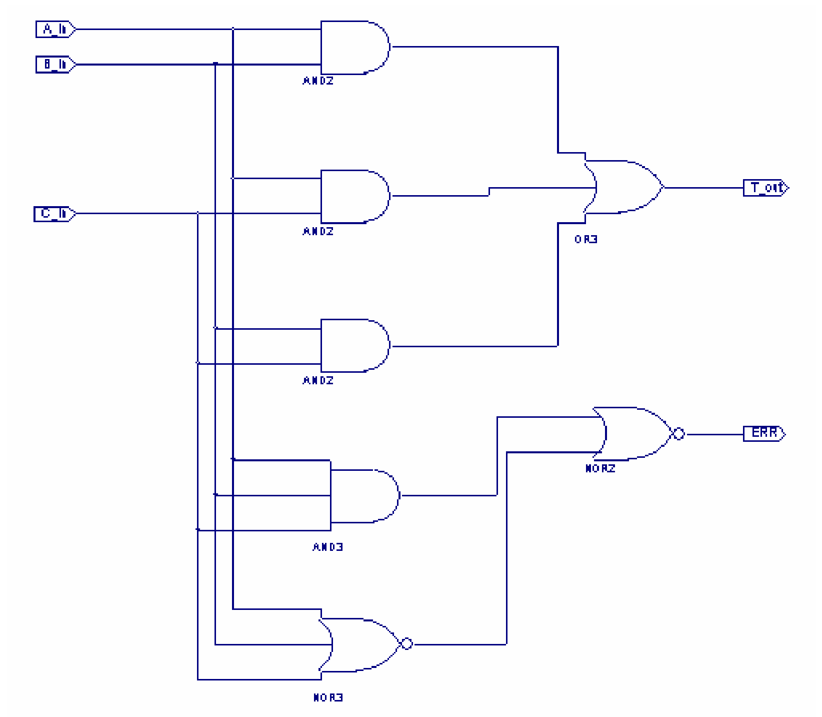


Figure 14. Schematic of Simple Voter with Error Signal

One final expansion is made to the Voter circuit used in the Pix TMR architecture. Additional logic is added to the circuit in Figure 15 to provide an indication of which input did not match the other two. Table 5 shows a truth table for the Voter circuit with four outputs. One is the voted output representing the input in majority, and the other three give indication of which input contained the mismatch. Figure 16 provides a schematic of this Voter circuit.

A	B	C	T_OUT	A_ERR	B_ERR	C_ERR
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	1	0
0	1	1	1	1	0	0
1	0	0	0	1	0	0
1	0	1	1	0	1	0
1	1	1	1	0	0	0

Table 4. Truth Table for Voter with Input Error Indication

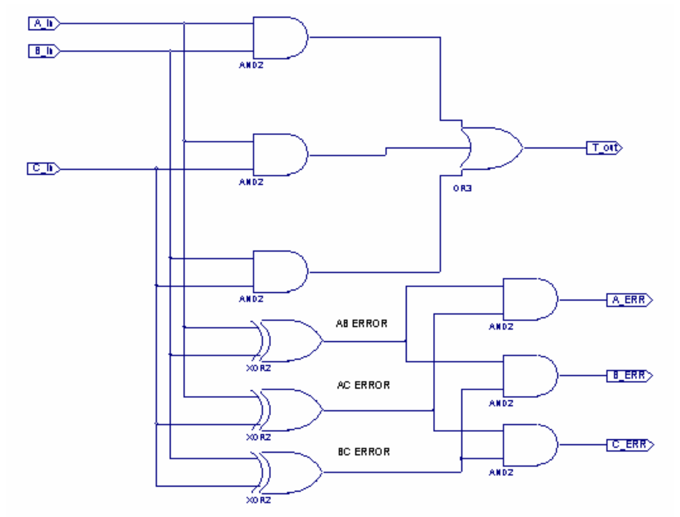


Figure 15. Schematic of Voter with Error Input Indication

The following code is a VHDL implementation of the one bit Voter shown in Figure 16. Lines 1 through 9 equate to the three XOR gates of the schematic. The remaining lines equate to the other AND gates and apply the corresponding signals to the outputs. Two new outputs are developed. The first specifies which bit of a multi-bit voter contains the error. This section of code is from bit 0. The Voter_err output is connected using a multi-input OR gate to the other Voter_err outputs from the other bits in the voter in order to give an indication that somewhere in the voter an error was detected.

```
1.    if A_in(0) /= B_in(0) then
2.            AB_err(0) := '1';
3.    end if;
4.    if A_in(0) /= C_in(0) then
5.            AC_err(0) := '1';
6.    end if;
7.    if B_in(0) /= C_in(0) then
8.            BC_err(0) := '1';
9.    end if;
10.   if (AB_err(0) = '1' AND AC_err(0) = '1') then
11.           Voter_err <='1';
12.           A_err  <= '1';
13.           Err(0) <= '1';
14.           True_out(0) <= B_in(0);
15.   elsif (AB_err(0) = '1' AND BC_err(0) = '1') then
16.           Voter_err <='1';
17.           B_err  <= '1';
18.           Err(0) <= '1';
19.           True_out(0) <= A_in(0);
20.   elsif (BC_err(0) = '1' AND AC_err(0) = '1') then
21.           Voter_err <='1';
22.           C_err  <= '1';
23.           Err(0) <= '1';
24.           True_out(0) <= B_in(0);
25.   else
26.           True_out(0) <= B_in(0);
27.   end if;
```

B. VOTING IN THE PIX – TMR ARCHITECTURE

1. Original TMR Design

The TMR architecture proposed by Lashomb [1] is depicted in Figure 17. The voter system reads the output of each processor during operations that interact with memory. Signals that are monitored by the voter are the PC, Write and Read commands, data address (for memory reads and writes) and the data itself for memory writes.

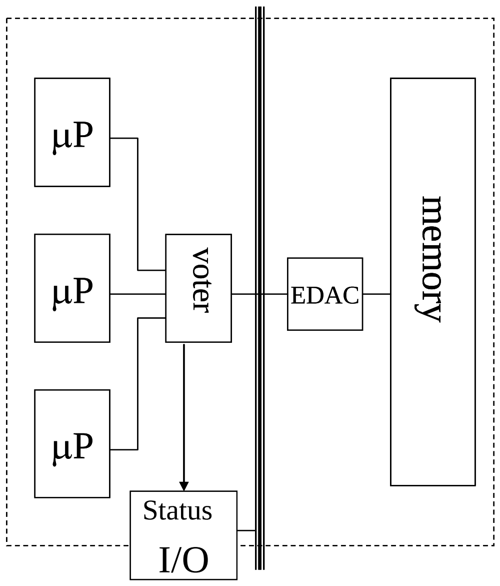


Figure 16. Traditional TMR Architecture (From [1])

In the event of an error, the processors would all simultaneously enter an Interrupt Service Routine which contained instructions for the processors to store each of the registers in the register file and then load the values back into the original registers. The operation of storing the values to memory utilized the voting circuits so that corrected data was used. At the end of the ISR, a jump instruction was inserted which allowed the PC to resume program fetching at the point the error was detected The event of detecting an error also triggered the generation of an *Error Syndrome*, which stored the PC value when the error was detected, the data address and a sequence of 102 bits from which the faulty processor and bit was determined.

2. Pix Voting System

a. Internal Voters

The TMR architecture designed in this thesis deviated from the design proposed in previous work. Instead of placing the voters *externally*, limiting their actions to accesses to memory, the voting function with the Pix processor is distributed, with voters placed *internal* to the processors. Voters monitor the outputs of all internal phase registers before the signals are applied to the combinatorial logic included in the respective phase. The result is that if an error occurs to the data in an internal phase register, the affect is immediately rectified, as corrected data is applied to the rest of the circuit via a voter. A direct benefit is that the processor does not have to enter an interrupt service routine in order to return to proper operation. The exception is if a fault occurs to a value stored in the register file in one of the processors. Although a voter would correct the value as it is forwarded to the EX phase, the original data stored in the register would still be faulty. The chance that an error could occur to the same bit of the same register in two different processors would increase. This could eventually violate the premise for using the voter circuit, which is that the value in the majority is assumed to be correct. A circuit that would stall the processor and immediately write the correct value back to the register could be added. Figure 17 shows a block diagram of the placement of the voters.

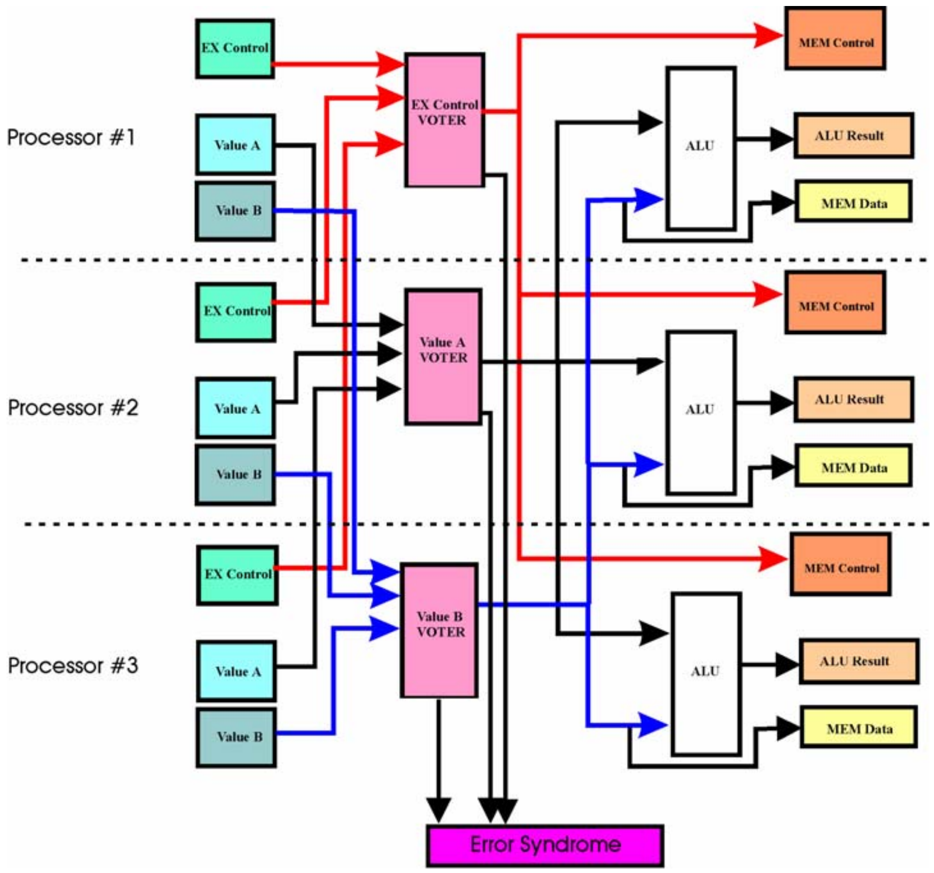


Figure 17. Triple Pix Processors with Internal Voters

A voter takes the same inputs from each phase register and determines what the correct out should be. This output is them applied to the combinatorial logic of the next stage. In Figure 17 the verified values from registers Value A and Value B are voted, and the output is applied to the ALU. If an error is detected by a voter, an Error Syndrome is also produced. Table 5 shows the total number of bits that are voted after each phase of the processor. As shown, the complete Pix TMR architecture votes 984 bits.

Register	Bits
IF/ID	64
ID/EX	120
EX/MEM	73
MEM/WB	71
Total (1) Pix	328
Total (3) Pix	984

Table 5. Voted Bits of each Processor Phase

b. Error Syndrome Decoding

In order to facilitate communicating the Error Syndrome between the Experiment FPGA and the Control FPGA on the CFTP circuit board, the Error Syndrome was limited to 32 bits (the width of the data bus).

	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Pix			STAGE				VOTER												BIT									
ID STAGE	A	B	C	0	0	0	1	■	■	10	9	8	7	6	5	4	3	2	1	9	8	7	6	5	4	3	2	1	0
EX STAGE	A	B	C	0	0	1	0	12	11	10	9	8	7	6	5	4	3	2	1	9	8	7	6	5	4	3	2	1	0
MEM STAGE	A	B	C	0	1	0	0	■	■	■	■	8	7	6	5	4	3	2	1	9	8	7	6	5	4	3	2	1	0
WB STAGE	A	B	C	1	0	0	0	■	■	■	■	8	7	6	5	4	3	2	1	■	8	7	6	5	4	3	2	1	0

Table 6. Error Syndrome Decoding

The coding that was implemented allowed the Error Syndrome to be reduced to 29 bits. Bits 28 through 26 represent the processor that contains the error. The next four bits, 25 through 22, indicate the phase in the processor in which the error resides. Finally, bits 21 through zero indicate which bit of the phase is in error. The largest number of voted bits in one phase is from the output of the ID/EX phase (120). In order to differentiate all 120 bits, the vector is first divided into individual multi-bit voters. To identify a specific bit, the Error Syndrome needs to specify the voter that contains the error and the bit number in that voter. The total number of bits that can be represented is then equal to the product of the number of voters and the number of bits. For the EX phase, there are 12 voters with each voter monitoring 10 bits, so all 120 bits of the phase are covered

c. FPGA Dependent TMR.

As discussed in [2], there are two major categories of FPGAs. The first type use anti-fuses to configure the circuit on the FPGA. Once programmed, the FPGA can not be reconfigured. The second type is Static Random Access Memory (SRAM) based FPGA. In these types of FPGA's the configuration is stored as bits in SRAM cells with determine the detailed inner-connections of the FPGA components.

(1) Anti-fuse FPGA TMR. Since the configuration of an anti-fuse FPGA permanently and physically alters the device, the configuration is not susceptible to SEU's. The most vulnerable part of the FPGA is the data stored in the internal registers, so the architecture developed in this thesis with only the internal phase registers triplicated and voted will provide adequate reliability. Figure 18 shows a schematic of a full TMR architecture where internal registers are triplicated with their outputs applied to voting circuits so that verified signals are provided to the rest of the circuit.

(2) SRAM FPGA TMR Since the configuration of an SRAM FPGA is stored as bits in memory cells, *the configuration itself* is susceptible to SEUs. This makes both the data of internal registers and the actual configuration vulnerable. Additionally, since up to 91% of the memory elements on the chip store the configuration, it is much more probable to have an SEU occur on a configuration bit [7]. To provide adequate reliability, all components implemented on an SRAM FPGA must be triplicated. Figure 19 shows a schematic where all components (except for voters) are triplicated. The output of each register is applied to voters. The output of the voter is then applied to three copies of the combinatorial logic in each phase. Each copy of the logic then is the input into the next phase register. Since SRAM FPGAs are used on the CFTP circuit board, the full TMR implementation as shown in Figure 19 used in this thesis. An architecture where the voters are also triplicated will be discussed in the future work section of this thesis.

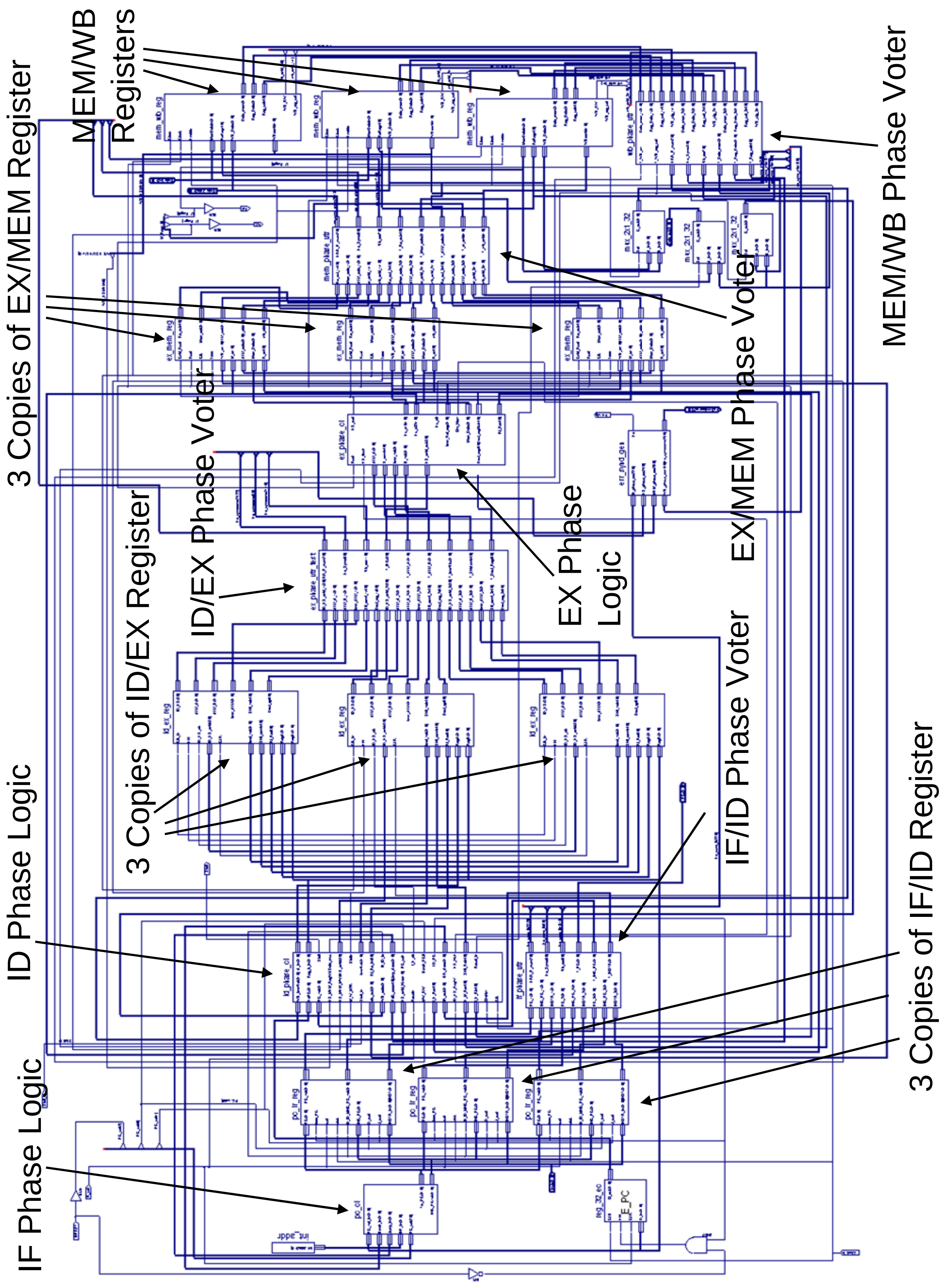


Figure 18. Schematic of TMR Internal Phase Registers

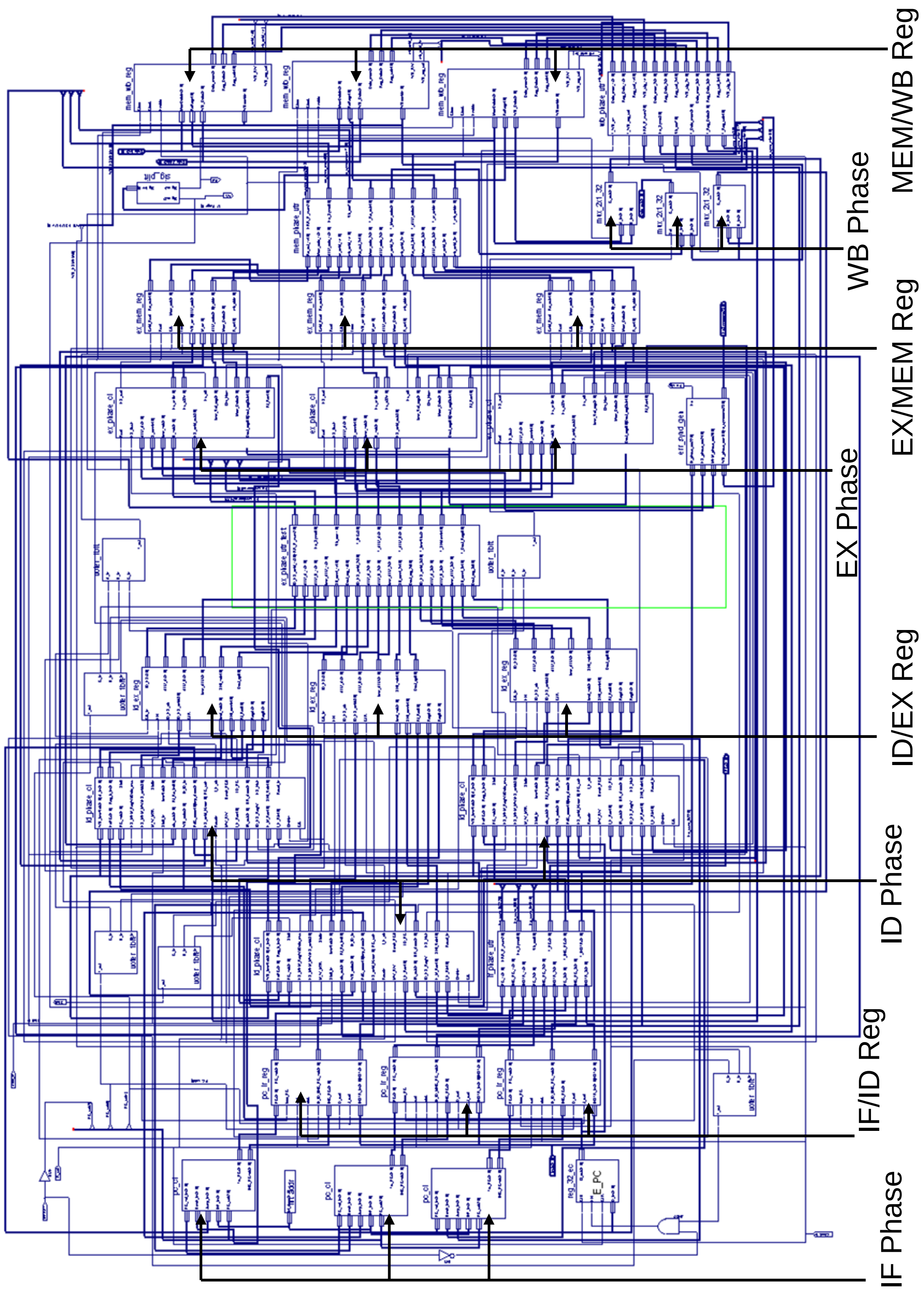


Figure 19. Schematic of Pix with Full TMR

C. CHAPTER SUMMARY

This chapter has described a simple voting circuit and then expanded the functionality to the level that is used in the Pix Processor. Additionally, the architecture of distributing the voting function to the output of the internal processor registers was detailed. Finally, two separate architectures were illustrated. The first provides adequate reliability for an anti-fuse based FPGA where the configuration is not susceptible to SEUs and a second, where all of the internal components are triplicated in order to provide added reliability to the complete FPGA configuration.

The next chapter describes the complete circuit implemented on the Experiment FPGA of the CFTP circuit board. The components include the full Pix TMR architecture, a triplicated memory block and a multiplexer to switch between outputting memory words and Error Syndromes to the Control FPGA. Additionally, the circuit implemented on the Control FPGA to interface with the Experiment FPGA will be described.

IV. THE COMPLETE TMR SYSTEM

A. PIX – TMR PROCESSOR

The complete TMR architecture utilizing the Pix processor illustrated previously in Figure 19 was reduced to a single schematic symbol and is shown in Figure 20.

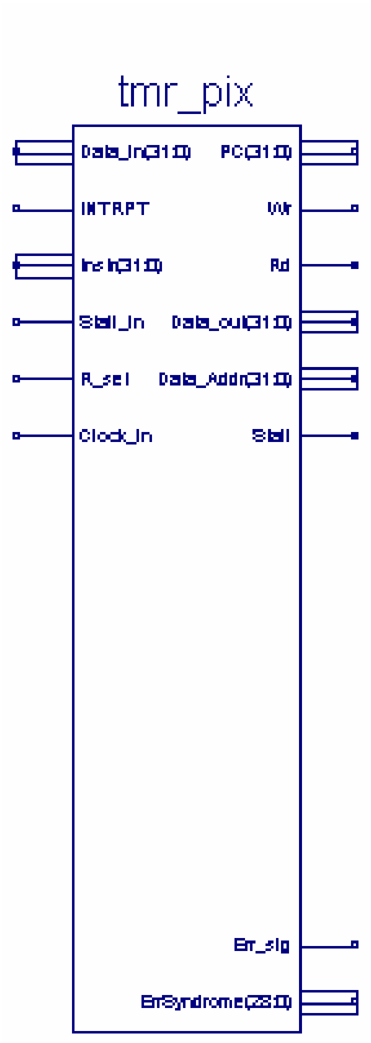


Figure 20. Schematic of Pix – TMR

Table 7 lists all the input and output signals for the unit.

Signal Name	Direction	Width (bits)	Description
Data_in	Input	32	Data fetched fom Data Memory
INTRPT	Input	1	External Inturrpt signal
Instr	Input	32	Instruction fetched from Instruction Memory
Stall_in	Input	1	External Stall signal
R_set	Input	1	External Reset signal
Clock_in	Input	1	Clock signal
PC	Output	32	Address to Instruction Memory
WR	Output	1	Write enable signal to Data Memory
Rd	Output	1	Read enable signal to Data Memory
Data_out	Output	32	Data to be stored in Data Memery
Data_Addr	Output	32	Address to Data Memory
Stall	Output	1	Indication that processor is stalled
Err_sig	Output	1	Indication that an Error was detected
ErrSyndrome	Output	29	Error Syndrome

Table 7. Signals of Pix – TMR

B. COMPLETE EXPERIMENT FPGA CIRCUIT

After completing the Pix-TMR architecture, the goal was expanded to design a complete system so that the triplicated processor system could be loaded on the Experiment FPGA, have its operation verified and the resilience of the TMR architecture in a radiation environment finally tested. Figure 21 shows the complete circuit programmed on the Experiment FPGA of the CFTP circuit board. It includes the Pix-TMR unit shown in Figure 20 and a memory unit which includes instructions for the Pix processors to execute (described in more detail in the next section).

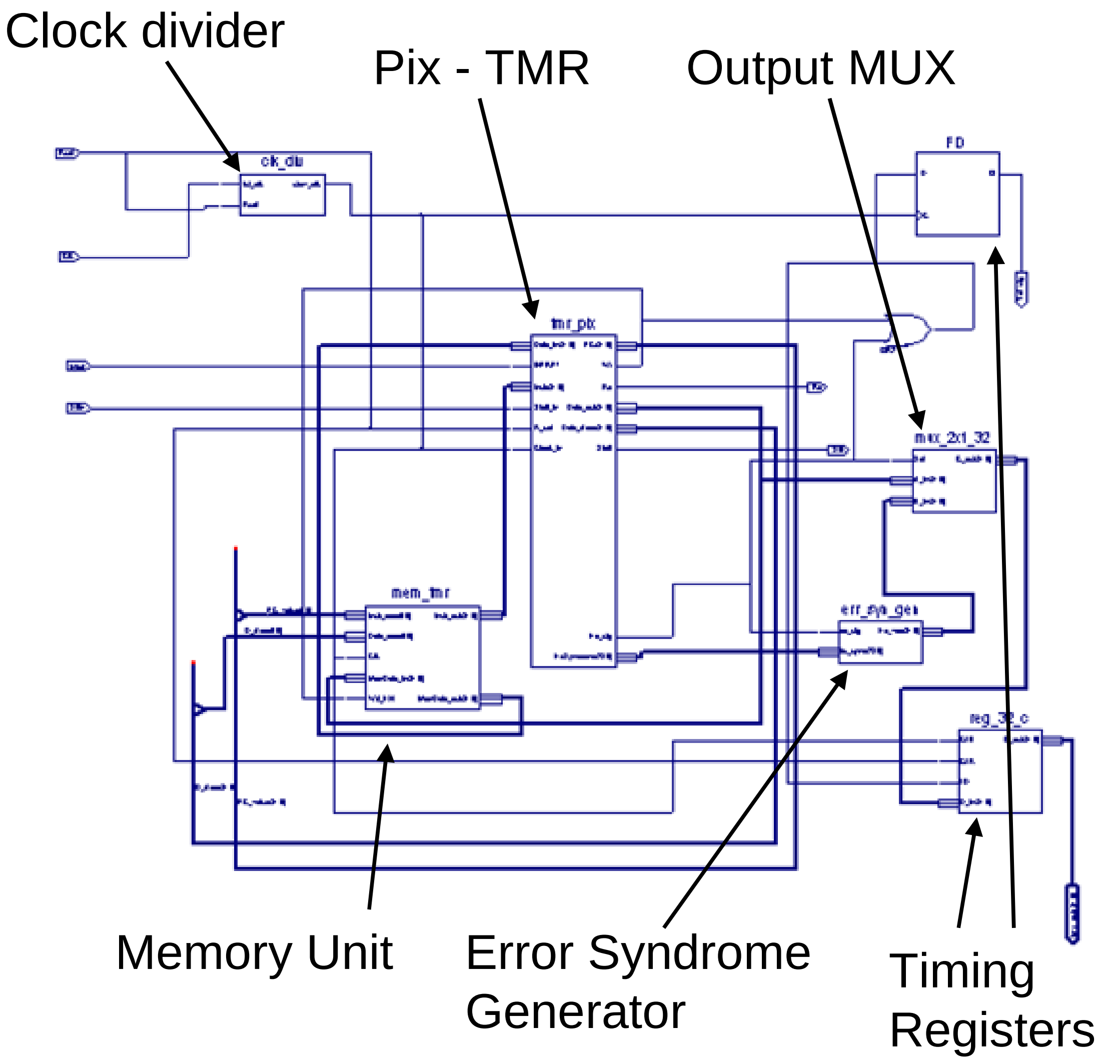


Figure 21. Complete Experiment FPGA Circuit

The schematic includes a clock divider used to divide the clock from 50 MHz, which is the CFTP circuit board oscillator frequency, by 2, resulting in a 25 MHz clocking signal. The circuit was designed to output two pieces of data, each consisting of 32 bits. The first is the data being written to memory by a *Store Word* instruction and the other an Error Syndrome developed when an error was detected. Due to the limited number of connections between the Experiment and Control FPGAs, these two signals shared the same connection pins by utilizing an output MUX. The Err_sig signal which indicated that a valid Error Syndrome exists, switched the MUX so that the Error Syndrome is outputted off the Experiment FPGA. This implementation also provided a priority to the

Error Syndrome such that if the processor is writing a value to memory (Wr signal asserted) at the same time the Err_sig signal is asserted, the Error Syndrome is applied to the output of the FPGA. The Error Syndrome was expanded from 29 to 32 bits by a unit labeled the *Error Syndrome Generator* which concatenates three *ones* to the most-significant bits of the Error Syndrome. This aids in the differentiation of the Error Syndrome from Memory data. The Experiment FPGA indicates the presence of valid data (memory data or Error Syndrome) to the Control FPGA through the Val_sig signal. It is asserted by either the Wr signal *or* the Err_sig signal. Finally, proper signal synchronization between the FPGA's is maintained by storing all outgoing bits in a register in the Experiment FPGA. This ensures that the signals are valid for one complete clock cycle, giving the Control FPGA an opportunity to latch the data. Table 8 lists all the input and output signals for the Experiment FPGA circuit.

Signal Name	Direction	Width (bits)	Description
Rset	Input	1	External Reset
Clk	Input	1	Clock signal
Intrpt	Input	1	External Interrupt signal
StlIn	Input	1	External stall signal
Val_sig	Output	1	Indication of valid data on output bus
Rd	Output	1	Processor executing read process with memory
Stll	Output	1	Indication that processor is stalled
tmr_out	Output	32	Either memory data or Error Syndrome

Table 8. Experiment FPGA Signals.

C. MEMORY UNIT

As discussed previously, the Harvard architecture of the pipelined Pix processor needs to accomplish two independent memory accesses in a single clock period. To accomplish this, a Dual Port Block RAM memory module was implemented using the Core Gen utility in the Xilinx ISE design program. Each memory cell in the block RAM is 32-bits wide and the block contains 64 cells.

1. Test Program

The following is the test program stored in the Block RAM. The goal of the test program was to have the Pix processors execute every instruction in the instruction set and utilize every register in the register file in order to exercise every component in each processor. If radiation caused an upset to the FPGA data or configuration, a thorough test program would maximize the chance that the output would be affected.

```
 1  00000000  00000000     nop
 2  00000001  00002820     add r5, r0, r0
 3  00000002  00003020     add r6, r0, r0
 4  00000003               LOOP1:
 5  00000003  20017fff     addi r1, r0, 0x00007fff
 6  00000004  00011020     add r2, r1, r0
 7  00000005  00011820     add r3, r1, r0
 8  00000006  00022100     sll r4, r2, 4
 9  00000007  24050001     addiu r5, r0, 1
10  00000008  24c60001     addiu r6, r6, 1
11  00000009  00a63820     add r7, r6, r5
12  0000000a  00c74020     add r8, r7, r6
13  0000000b  00e84820     add r9, r8, r7
14  0000000c  00a95023     subu r10, r9, r5
15  0000000d  012a5825     or r11, r10, r9
16  0000000e  00ea6026     xor r12, r10, r7
17  0000000f  394dfff6     xori r13, r10, 0x0000fff6
18  00000010  00837027     nor r14, r3, r4
19  00000011  01c47824     and r15, r4, r14
20  00000012  20af0001     addi r15, r5, 1
21  00000013  11e0ffef     beq r15, r0, LOOP1
22  00000014  35d0ffff     ori r16, r14, 0x0000ffff
23  00000015  307103c0     andi r17, r3, 0x000003c0
24  00000016  00009020     add r18, r0, r0
25  00000017  1412ffeb     bne r0, r18, LOOP1
26  00000018  000e9d03     sra r19, r14, 20
27  00000019  000ea042     srl r20, r14, 1
28  0000001a  0640ffe8     bltz r18, LOOP1
29  0000001b  0184b804     sllv r23, r4, r12
30  0000001c  0170c006     srlv r24, r16, r11
31  0000001d  00aec82a     slt r25, r14, r5
32  0000001e  00aed02b     sltu r26, r14, r5
33  0000001f  2b3bffff     slti r27, r25, -1
34  00000020  2f3cffff     sltiu r28, r25, -1
35  00000021  0601ffe0     bgez r16, 2
36  00000022  008ee825     or r29, r14, r4
37  00000023  07a1ffde     bgez r29, 2
38  00000024  201e007f     addi r30, r0, 0x0000007f
39  00000025  001efe40     sll r31, r30, 25
40  00000026               LOOP2:
41  00000026  23ff0001     addi r31, r31, 1
42  00000027  17e0fffe     bne r31, r0, LOOP2
43  00000028  0400ffda     bltz r0, LOOP1
44  00000029  0410ffd9     bltzal r0, LOOP1
45  0000002a  ac06003e     sw r6, 62(r0)
46  0000002b  08000003     j LOOP1
```

Another function of the test program was to communicate that the processor was still operating normally by outputting a periodic and predictable value. This was accomplished by first clearing the contents of register R6 (line 3), and then incrementing the value in R6 (line 10) once during each program iteration. A four second delay was implemented in lines 38 through 42, which finally ended in storing the value in R6 to memory, simultaneously outputting the value from the Experiment FPGA. The last instruction resets the Program Counter to jump back to address 3, so the processors execute the program again. The net result was that at test commencement a value of zero was outputted off the CFTP board and every four seconds the observed value was incremented by one to prove continued processor operation.

2. Reliable Memory Unit

In order to add reliability to the memory unit, the RAM blocks implemented on the FPGA were also triplicated. A simple voting circuit (no error detection signal or Error Syndrome generation) was included. The outputs of the individual memory blocks were applied to a 32-bit voter to reduce the chance that a single fault would cause erroneous memory data to be applied to the processors. If there is not enough room on the FPGA to hold three copies of the memory block, more complex but spatially efficient memory error detection and corrections techniques could be used for example, Single Error Correction, Double Error Detection (SECDED) Hamming codes.

D. CONTROL FPGA

As discussed previously, the function of the Control FPGA is to communicate externally to the space craft Command and Data Handler (C&DH) utilizing a PC-104 bus interface and to control the Experiment FPGA, specifically to configure, initialize and receive data from it. The work covered by this thesis included the circuit implemented on the Control FPGA which interfaced with the Experiment FPGA. Figure 22 shows a flow diagram which describes the monitoring of signals and the flow of data.

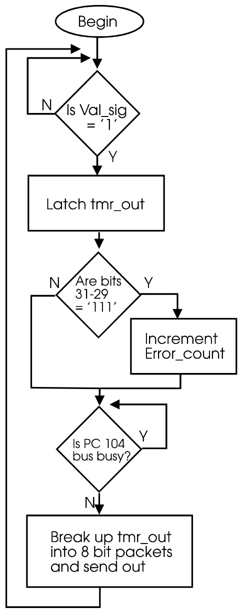


Figure 22. FPGA Interface Data Flow Chart

The additional functions that the Control FPGA performed were to read back the Experiment FPGA configuration every 30 seconds, compare it to the original configuration data stored in Flash memory and print out any discrepancies. Additionally, the Control FPGA reconfigured the Experiment FPGA if the *Error-counter* reached 128 errors [6].

E. CHAPTER SUMMARY

This chapter described the circuit implemented on the Experiment FPGA of the CFTP circuit board. The circuit consisted of the complete Pix TMR unit, a triplicated and voted memory unit and other components to exchange data with the Control FPGA. Additionally, the interface circuit implemented in the Control FPGA was described.

The next chapter provides an analysis of the test result data from the proton beam test performed at the University of California at Davis.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RADIATION TEST OF PIX-TMR

A. PROTON RADIATION TEST

On November 15, 2005 the CFTP team conducted proton radiation tests using the Isochronous Cyclotron at Crocker Nuclear Laboratory, University of California-Davis. Figure 23 was produced using the CFTP Error Visualization Tool (EVT) developed at Naval Postgraduate School [6]. The figure represents a small section of the test performed. The first item displayed in the figure is the physical location of the implemented Pix-TMR experiment circuit on the Virtex 2 FPGA. As described in [2], the FPGA is composed of Configurable Logic Blocks (CLBs) that contain Look-Up-Tables (LUTs), registers and reprogrammable routing controls (multiplexers). The CLBs are shown as small pink squares. The other circuit components shown in the diagram are the Programmable Interconnect Points (PIPs). These circuits provide the routing path used to connect CLBs and Input Output Buffers (IOB). These are represented in the figure as green plus-signs. The specific implementations and connections for all the CLBs and PIPs are maintained as configuration data bits for the FPGA. The figure only shows that CLBs and PIPs that are used for the Pix-TMR experiment implementation. All of the unused CLBs and PIPs have configuration data bits which are equal to zero. From the figure, it can be estimated that the implemented circuit utilizes approximately 25% of the FPGA hardware. The other feature of the EVT is that induced faults are spatially shown on the FPGA implementation. The configuration data is organized by specific blocks, which allows the EVT to determine the location of the error. The figure illustrates that approximately half of the faults (23 out of 50) occurred in areas that did not contain configuration data for the experiment circuit. Additionally, since not all the components of a CLB or PIP are used in the circuit implementation, even a fault occurring inside a used CLB can have no effect to circuit operation.

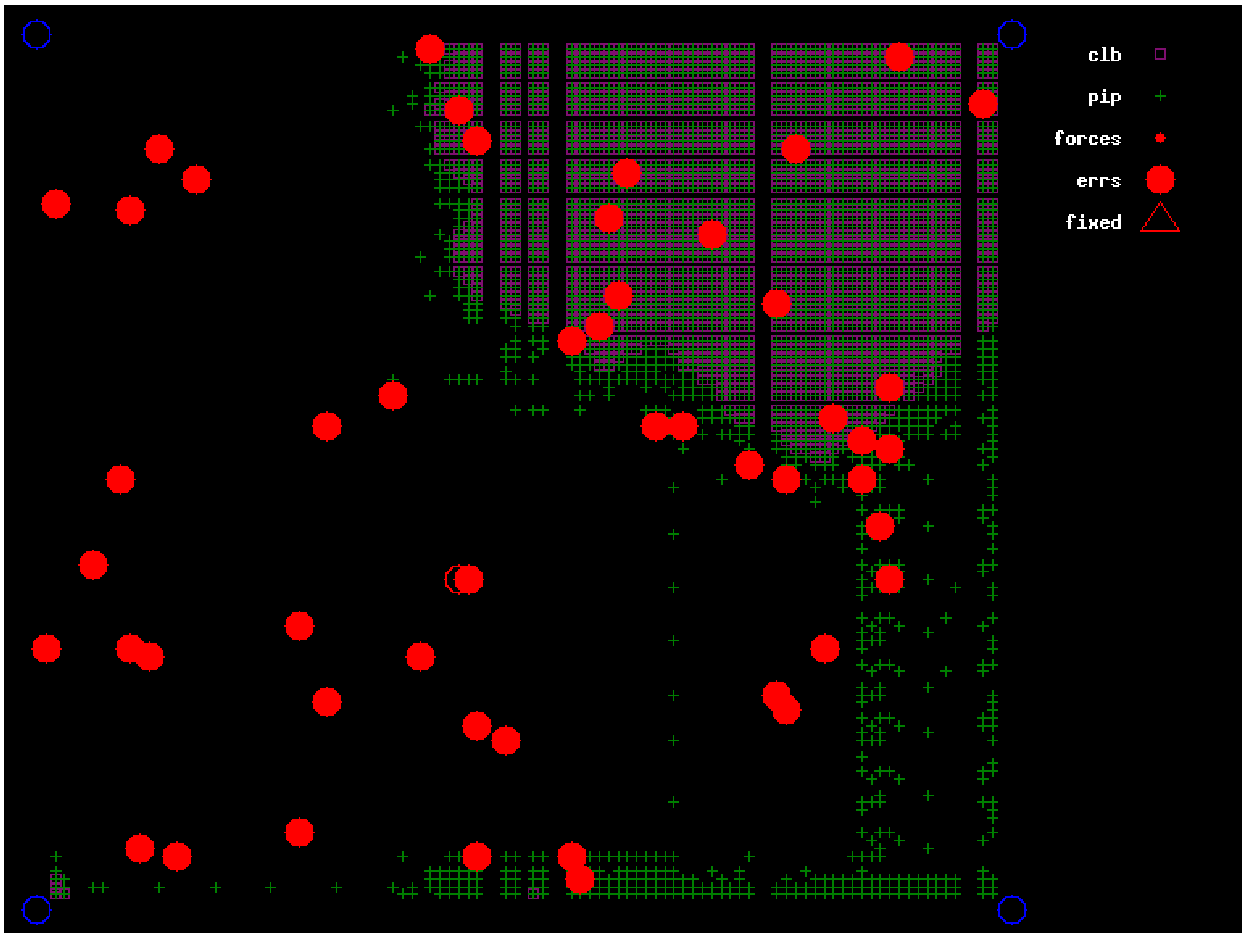


Figure 23. Pix-TMR FPGA Test Visualization

B. EXPECTED RESULTS

Based on the design of the TMR system and the implementation of the circuit on the FPGA, potential fault categories were formed based on their predicted effect on circuit operation.

1. No Effect Faults

With the test program initialized and proper circuit operation verified (incrementing output counter), the only way to verify that a fault occurs is by comparing actual configuration data to a copy stored in Flash memory. These faults can occur in areas where the CLBs are not part of the implementation. Another possible cause is an error to the configuration data for an unused component inside a utilized CLB.

2. Total Failure Faults

After initial circuit operation is verified, a fault occurs that renders the experiment circuit completely inoperable. If the Val_sig output is never asserted, no data from the

Experiment FPGA will be printed to the screen. This condition would be caused by a configuration data fault to a circuit component that is not protected through triplication, or any other mitigation technique. These elements are also known as *single points of failure*. An example would be a fault to the configuration circuitry linking the output of a voter to the next stage of the processor. In this case, all three copies of the combinatorial logic would get identical *faulty* data. To further the example, if the data pertaining to the Instruction Register were permanently altered (voter output configuration changed), the processor might continuously assume invalid instructions and insert NOPs in their place.

3. TMR Mitigated Faults

After initial circuit operation is verified, a fault occurs in one copy of a triplicated circuit. The faulty data is voted out as the operation of the processor continues uninterrupted. Simultaneously, an error report is generated which specifies the mismatching processor, phase and bit.

C. TEST DATA REPORTS ANALYSIS

The output format from the CFTP circuit board consisted of two lines per report. The first consists of a time indication including hour, minute and second. The second line lists the error counter value (Errcnt) and either the data being written to memory (mem data: *incrementing value*) or an Error Syndrome (signaled with ****ERROR*).

1. Test Data Section A

The following is a portion of the test output during the radiation test. The first output is generated at time 22:07:09. Due to an unexplained flaw, the error counter initialized to *1*. Lines 1 through 4 of the output shows that the four-second iteration of the instructions loaded in the block RAM is functioning properly. Lines six and eight report the Error Syndrome: e4808008. This decodes to: Processor C, EX phase, bit 12 of the ALU B input. Both of these reports occur during the time 22:07:13. The operation of the experiment continues correctly as verified by the correct memory data output in line 10. This pattern of two Error Syndromes continues for the next eight seconds, but the number being written to memory (incremented counter) is incorrect. Curiously, in line 28 the incremented data being sent to memory regains proper sequence based on the original timing and data value. At time 22:07:25, the Error Syndrome: e8a3c3ff is reported. This decodes to Processor B, EX Phase, but includes errors in two voters. The mismatched bits could belong to Destination Register, bits 9-0, or ALU B input, bits 31-0, or some combination of both. The precise source of the bits in error is lost due to the compression caused by encoding 984 bits into 29. Additionally, the fact that several different bits are mismatched is a unique manifestation of configuration bit errors. A single bit error in the configuration data does equate to just a single bit error in the processor data. The number of bits in error is also dependent on the data being processed. For example, if the configuration error disabled a 32-bit register, the attempted loading of value ffffffff (hex) would result 32 bit errors, while loading of value 00000001 (hex) would result in a single bit error.

```
1.          22:07:09
2.          Errcnt: 01    mem data: 00000001
3.          22:07:13
4.          Errcnt: 01    mem data: 00000002
5.          22:07:13
6.          ***ERROR: Errcnt: 01    mem data: e4808008
7.          22:07:13
8.          ***ERROR: Errcnt: 02    mem data: e4808008
9.          22:07:17
10.         Errcnt: 03    mem data: 00000003
11.         22:07:17
12.         ***ERROR: Errcnt: 03    mem data: e4808008
13.         22:07:17
14.         ***ERROR: Errcnt: 04    mem data: e4808008
15.         22:07:21
16.         Errcnt: 05    mem data: 00000003
17.         22:07:21
18.         ***ERROR: Errcnt: 05    mem data: e4808008
19.         22:07:21
20.         ***ERROR: Errcnt: 06    mem data: e4808008
21.         22:07:25
22.         Errcnt: 07    mem data: 00000003
23.         22:07:25
24.         ***ERROR: Errcnt: 07    mem data: e8a3c3ff
25.         22:07:26
26.         Errcnt: 08    mem data: 00000004
27.         22:07:30
28.         Errcnt: 08    mem data: 00000007
29.         22:07:30
30.         ***ERROR: 01 Errcnt: 08    mem data: e8a3c3ff
31.         22:07:34
32.         Errcnt: 09    mem data: 00000008
33.         22:07:34
34.         ***ERROR: 01 Errcnt: 09    mem data: e8a3c3ff
35.         22:07:38
36.         Errcnt: 0a    mem data: 00000009
37.         22:07:38
38.         ***ERROR: 01 Errcnt: 0a    mem data: e8a3c3ff
39.         22:07:42
40.         Errcnt: 0b    mem data: 0000000a
41.         222:07:42
42.         ***ERROR: 01 Errcnt: 0b    mem data: e8a3c3ff
43.         22:07:46
44.         Errcnt: 0c    mem data: 0000000b
45.         22:07:46
46.         ***ERROR: 01 Errcnt: 0c    mem data: e8a3c3ff
47.         22:07:50
48.         Errcnt: 0d    mem data: 0000000c
49          .22:07:50
50.         ***ERROR: 01 Errcnt: 0d    mem data: e8a3c3ff
```

```
51.         22:07:50
52.         Errcnt: 0e    mem data: 0000000d
53.         22:07:54
54.         Errcnt: 0e    mem data: 00000010
```

2. Test Data Section B

The configuration errors reported in the previous section fall into the category of TMR mitigated faults, in that the processors continued to operate (with some output errors). The next section of data results also fall into this category, but operation of the circuit cannot be verified because errors were being discovered too frequently (possibly every clock cycle). The operation of the voters and the output circuit implies that the system on the FPGA is still functional. Lines 1 through 34 show the last 16 errors reported before the Control FPGA initiated a reconfiguration. All the reports shown below take place within a single second (time 22:06:42). If the reports generated at time 22:06:41 (one second earlier) were included, a total of 51 errors were reported in less than two seconds.

```
1.          22:06:42
2.          ***ERROR: 01 Errcnt: 6f    mem data: e0800080
3.          22:06:42
4.          ***ERROR: 01 Errcnt: 70    mem data: e0800080
5.          22:06:42
6.          ***ERROR: 01 Errcnt: 71    mem data: f0807bff
7.          22:06:42
8.          ***ERROR: 01 Errcnt: 72    mem data: f0807bff
9.          22:06:42
10.         ***ERROR: 01 Errcnt: 73    mem data: f0807bff
11.         22:06:42
12.         ***ERROR: 01 Errcnt: 74    mem data: e0800080
13.         22:06:42
14.         ***ERROR: 01 Errcnt: 75    mem data: e0800080
15.         22:06:42
16.         ***ERROR: 01 Errcnt: 76    mem data: e0800080
17.         22:06:42
18.         ***ERROR: 01 Errcnt: 77    mem data: e0800080
19.         22:06:42
20.         ***ERROR: 01 Errcnt: 78    mem data: e0800080
21.         22:06:42
22.         ***ERROR: 01 Errcnt: 79    mem data: f08063f1
23.         22:06:42
24.         ***ERROR: 01 Errcnt: 7a    mem data: e0800080
25.         22:06:42
26.         ***ERROR: 01 Errcnt: 7b    mem data: e0800080
27.         22:06:42
28.         ***ERROR: 01 Errcnt: 7c    mem data: e0800080
29.         22:06:42
30.         ***ERROR: 01 Errcnt: 7d    mem data: e0800080
31.         22:06:42
32.         ***ERROR: 01 Errcnt: 7e    mem data: e0800080
33.         22:06:42
34.         ***ERROR: 01 Errcnt: 7f    mem data: f0807bff
```

3. Test Data Section C

The final section of test data indicates a fault that has elements from two of the categories previously listed. Processor operation is verified by the output consisting of data being incremented and being written to memory, however, some values are missing. The operation also deviates from expected behavior because all of the output values occur at time 22:08:26, instead of every four seconds. The cause of this behavior may be from a configuration error to the clock divider circuit, with is not protected by any fault mitigation technique. The missing incrementing counter values (i.e. 0000001d, 00000020) were possibly lost due to a buffer overload situation in the PC-104 interface.

```
22:08:26
Errcnt: 16    mem data: 0000001c
22:08:26
Errcnt: 16    mem data: 0000001f
22:08:26
Errcnt: 16    mem data: 00000021
22:08:26
Errcnt: 16    mem data: 00000023
22:08:26
Errcnt: 16    mem data: 00000025
22:08:26
Errcnt: 16    mem data: 00000028
22:08:26
Errcnt: 16    mem data: 0000002c
```

4. Test Data Section D

Faults pertaining to the second category (total system failure) were also observed. The only indication on the screen was the configuration data comparison performed every 30 seconds (controlled by the Control FPGA). In between the comparison there was no output of either memory data or Error Syndrome.

D. TEST DATA ANALYSIS

In total, the test data indicates that a total of 518 faults were induced by radiation. Of those faults, 64 affected CLBs utilized by the circuit configuration (12.4%). Eighteen unique Error Syndromes were generated by the voting system and at least three more errors affected circuit operation by changing the output behavior without producing an error report (4.1%).

As indicated by the percentages, the *No Effect Fault* was most prevalent. This is a function of the size of the circuit implemented. As the circuit is modified and expanded, this percentage will go down.

The goal of the CFTP research is to reduce the number of *Total Failure Faults* by converting them to *Mitigated Faults*, that is, have the error reported, but the circuit operation continues unhampered. The limiting factors in the test were the way the error reports were generated and then reported. A change to the error generating system so that a report is outputted only if it is *unique* would have prevented an immediate FPGA reconfiguration because the same error is causing an error report every clock cycle.

E. CHAPTER SUMMARY

This chapter described the results of the radiation test performed at the Crocker Nuclear Laboratory, University of California-Davis. Categories of faults were listed based on their affect on system operation. Examples were then given of the output data from the Pix-TMR Experiment test and a fault analysis.

The next chapter provides a conclusion to the thesis and a recommendation for continued work.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

This thesis described the design of the Pix processor, a five-stage, pipelined RISC micro processor. The architecture of a Triple Modular Redundant system utilizing three Pix processors and voter circuitry was then developed. Finally, Results from a radiation test were analyzed and discussed.

A. SUMMARY

The Pix processor is a pipelined RISC processor based on the MIPS architecture developed by Hennessy and Patterson [5]. Its data and address busses are both 32 bits wide. It includes data forwarding to reduce the need to stall the processor when a data hazard could potentially exist due to the pipelining process. Thirty-seven instructions are implemented for the processor that include 12 arithmetic, 11 logical, eight branch, four jump and two memory access. The processor also has the ability to jump to a pre-loaded Interrupt Address and then return back to the last PC value upon an external Interrupt initiation.

The Pix Processor is the central component to the TMR architecture developed in this thesis. Instead of voting the output of the processor, a distributed voting system was developed where voters are placed inside the processor, on the output of each of the phase registers. This enables reliable processor operation (radiation induced faults mitigated) without having to stall the processor to verify and correct the data in all the internal registers. Additionally this voting system produces an Error Syndrome which gives additional information pertaining to the fault.

Finally, the thesis describes the first radiation test performed on a TMR processor system by the NPS CFTP research program. Effects of SEUs on circuit operation were described that included unaffected circuit operation (SEU to configuration data not used for the implemented circuit), mitigated faults (TMR architecture corrected the fault), and faults that affected processor output ranging from slight output deviation to complete system failure.

B. CONCLUSIONS

The design developed in this thesis reached or surpassed all the goals set at the beginning of the research. A microprocessor was developed that expanded the functionality of the KDLX processor used in the earlier stages of the CFTP research program. The Pix processor was developed in a modular format which facilitates future modifications and expansions. Additionally, an open source assembler for the DLX processor was modified to convert standard assembly language to Pix machine code [6].

The TMR architecture changed the original design of voting processor external outputs to a distributed function where the voting is accomplished at the processor internal phase registers. This is an effective technique because errors are corrected as data is being processed in the microprocessor.

The results from the radiation test gave insight to the relationship between configuration bit faults and their effect on the implemented system operation. As discussed previously, a single configuration bit error can have different manifestations in the processor output. An additional benefit of the voter system is that the source of the fault can be more easily ascertained (though still difficult), since the phase that the fault occurred in can be deduced.

C. FOLLOW-ON RESEARCH

Each component of the Pix-TMR system can be expanded. First the Pix processor can be improved. A math co-processor and instructions for floating point operations could be added. Additionally supervisory mode instructions can be implemented to facilitate interfacing the processor with an operating system.

The voting circuit and specifically the Error Syndrome generation can be improved to give better data on the location of the originating SEU. A system where Error Syndromes can be stored on the FPGA, for example, a block RAM could be developed so that all reports are saved, and not lost due to a slow PC-104 interface. This effort can be expanded to include changing the code on the Control FPGA that interfaces with the Experiment FPGA.

As noted in the previous chapter, single points of failure still exist in the Pix-TMR architecture. These weak points should be eliminated as much as possible. One method is described in [8] where the voters are also triplicated, with the output of each voter feeding only a single copy of the subsequent phase logic(also triplicated), as shown in Figure 24. If one voter is compromised due to a configuration bit error, Correct data is still outputted to the other copies of the logic. After the next phase of voting, all three copies of logic will once again contain correct data (assuming no other faults occur).

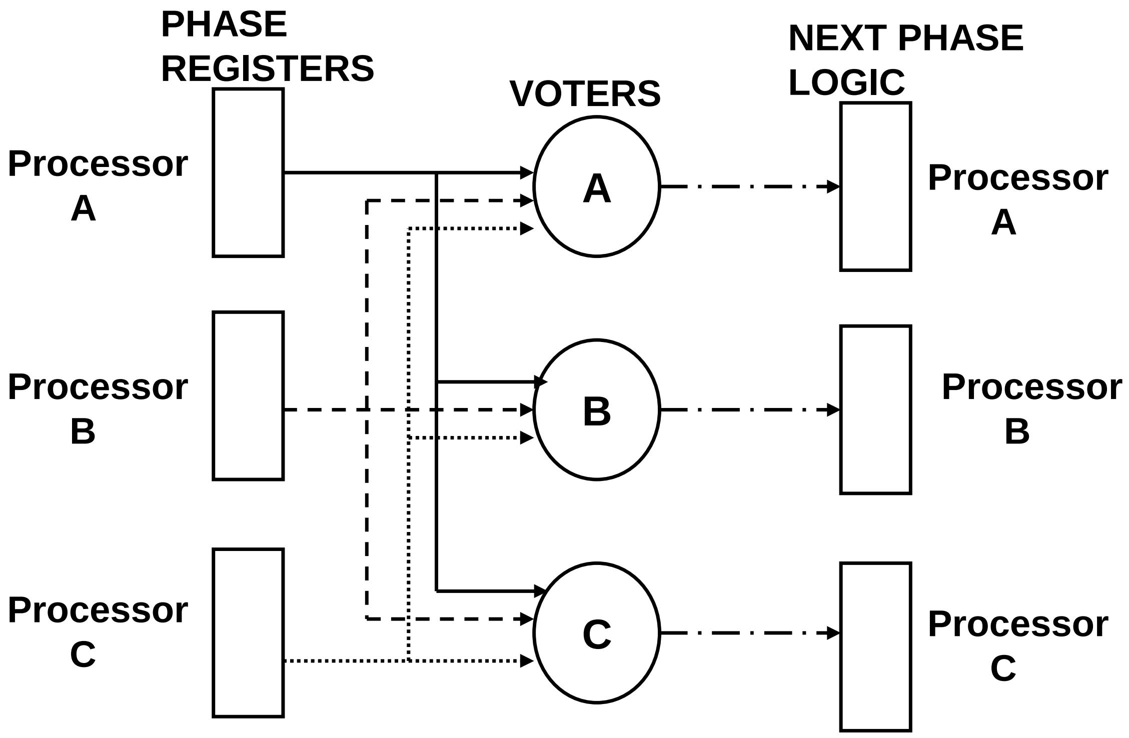


Figure 24. TMR With Triplicated Voters (After: [8])

Finally, continued testing with the CFTP Error Visualization and Injection Tool and radiation tests needs to be performed to fully observe and understand the effect of configuration bit errors on the operation of the circuit implemented on an FPGA. Mitigation of all faults that would cause complete system failure is extremely difficult. A process where the Control FPGA can detect a system failure (possibly using a watch-dog timer) must be developed to initiate a system reconfiguration. This system should also contain a feature which determines the last known state (i.e PC address) of the processor and reinitialize it to continue from that state.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: PIX INSTRUCTION SET

This appendix lists all of the instructions implemented for the Pix processor.

Instruction	Format	Syntax	Desciption
ADD	R	ADD rd, rs, rt	Add word: The word value in general register *rt* is added to the word value in general register *rs* and the result is placed into general register *rd* . If the addition results in 32-bit 2's complement arithmetic overflow, the overflow flag is set.
ADDI	I	ADDI rt,rs, immediate	Add Immediate Word: The 16 bit *immediate* is sign -extended and added to the contents of the general register *rs* to form the result. The result is placed into general register *rt* .If the addition results in 32-bit 2's complement arithmetic overflow, the overflow flag is set.
ADDIU	I	ADDIU rt, rs, immediate	Add Immediate Unsigned Word: The 16 bit immediate is sign-extended and added to the contents of the general register *rs* to form the result. The result is placed into general register *rt* . No integer overflow excetption occurs under any circumstances.
ADDU	R	ADDU rd, rs, rt	Add unsigned word: The word value in general register *rt* is added to the word value in general register *rs* and the result is placed into general register *rd.* If the addition results in 32-bit 2's complement arithmetic overflow, the overflow flag is not set.
AND	R	AND rd, rs, rt	And: The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into register *rd.*
ANDI	I	ANDI rt, rs, immediate	The 16 bit immediate is zero-extended and combined with the contents of general register *rs* in a bit-wise logical AND operation. The result is placed into general register *rt* .
BEQ	I	BEQ rs, rt, offest	Branch On Equal: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . The contents of general registers *rs* and *rt* are compared. If the two registers are equal, then the program branches to the target address.
BGEZ	I	BGEZ rs, offset	Branch On Greater Than Or Equal To Zero: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . If the contents of general registers *rs* have the sign bit cleared, then the program branches to the target address.
BGEZAL	I	BGEZAL rs, offset	Branch On Greater Than Or Equal To Zero And Link: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . Unconditionally, the address of the incremented PC is placed in the link register, *r31* . If the contents of general registers *rs* have the sign bit cleared, then the program branches to the target address.
BGTZ	I	BGTZ rs, offset	Branch On Greater Than Zero: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address.
BLEZ	I	BLEZ rs, offset	Branch On Less Than Or Equal To Zero: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . If the contents of general register *rs* have the sign bit set or are equal to zero, then the program branches to the target address.
BLTZ	I	BLTZ rs, offest	Branch On Less Than Zero: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . If the contents of general register *rs* have the sign bit set, then the program branches to the target address.
BLTZAL	I	BLTZAL rs, offest	Branch On Less Than Zero And Link: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . Unconditionally, the address of the incremented PC is placed in the link register, *r31* . If the contents of general register *rs* have the sign bit set, then the program branches to the target address.
BNE	I	BNE rs, rt, offset	Branch On Not Equal: A branch target address is computed from the sum of the incremented PC value and the sign-extended 16-bit *offset* . The contents of general registers *rs* and *rt* are compared. If the two registers are not equal, then the program branches to the target address.
J	J	J target	Jump: The 26 bit target address is combined with the high-order 6 bits of the incremented PC. The program unconditionally jumps to this calculated address.
JAL	J	JAL target	Jump: The 26 bit target address is combined with the high-order 6 bits of the incremented PC. The program unconditionally jumps to this calculated address. The address of the incremented PC is placed in the link register, *r31* .
JALR	R	JALR rs; JALR rd,rs	Jump And Link Register: The program unconditionally jumps to the address contained in general register *rs.* The address of the incremented PC is placed in general register *rd* . The default value of *rd* , if omitted in the assembly language instruction is 31.
JR	R	JR rs	Jump Register: The program unconditionally jumps to the address contained in general register *rs.*

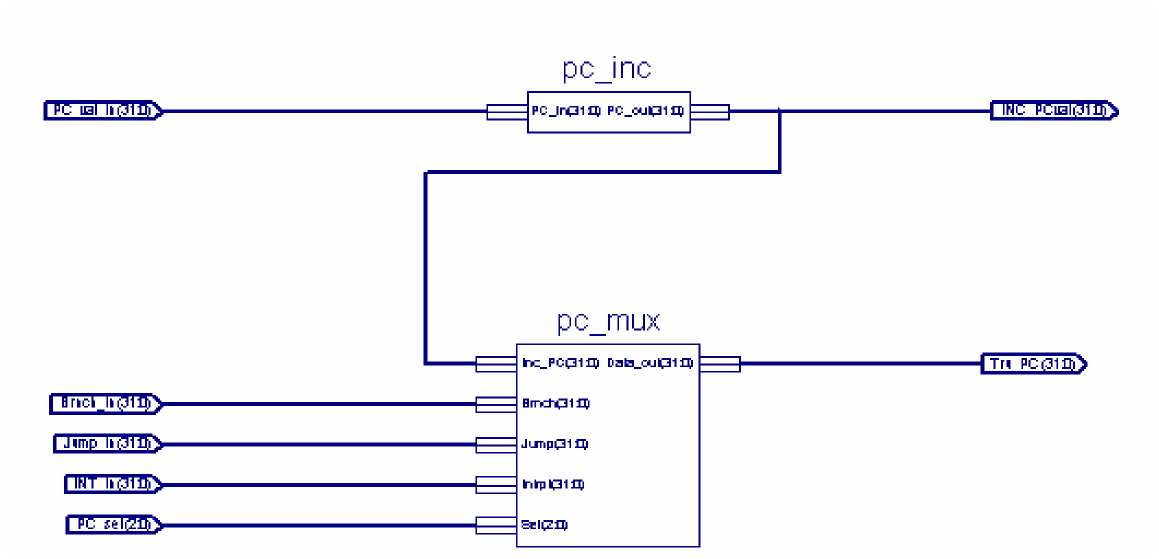
LW	I	LW rt, offset(base)	<u>Load Word</u>: The 16 bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual addres. The contents of the word at the memory location specified by the effective address are loaded into general register *r*.
NOR	R	NOR rd, rs, rt	<u>Nor</u>: The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical NOR operation. The result is placed into register *rd.*
OR	R	OR rd, rs, rt	<u>OR</u>: The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical OR operation. The result is placed into register *rd.*
ORI	I	ORI rt, rs, immediate	<u>OR Immediate:</u> The 16 bit immediate is zero-extended and combined with the contents of general register *rs* in a bit-wise logical OR operation. The result is placed into general register *rt* .
SLL	R	SLL rd, rt, sa	<u>Shift Word Left Logical:</u> The contents of the low-order word of general register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The word result is placed in register *rd* .
SLLV	R	SLLV rd, rt, rs	<u>Shift Word Left Logical Variable:</u> The contents of the low-order word of general register *rt* are shifted left by the low-order five bits contained in general register *rs* , inserting zeros into the low-order bits. The word result is placed in register *rd* .
SLT	R	SLT rd, rs, rt	<u>Set On Less Than:</u> The contents of general register *rt* are subtracted from the contents of general register *rs* . Considering both quantities as signed integers, if *rs* is less than the contents of *rt* , the result is set to one; otherwise the result is set to zero. The result (one or zero) is placed into general register *rd* .
SLTI	I	SLTI rt, rs, immediate	<u>Set On Less Than Immediate</u>: The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs* . Considering both quantities as signed integers, if *rs* is less than the sign-extended immediate, the result is set to one; otherwise the result is set to zero. The result (one or zero) is placed into general register *rt* .
SLTIU	I	SLTIU rt,rs, immediate	<u>Set On Less Than Immediate Unsigned</u>: The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs* . Considering both quantities as unsigned integers, if *rs* is less than the sign-extended immediate, the result is set to one; otherwise the result is set to zero. The result (one or zero) is placed into general register *rt* .
SLTU	R	SLTU rd, rs, rt	<u>Set On Less Than Unsigned:</u> The contents of general register *rt* are subtracted from the contents of general register *rs* . Considering both quantities as unsigned integers, if *rs* is less than the contents of *rt* , the result is set to one; otherwise the result is set to zero. The result (one or zero) is placed into general register *rd* .
SRA	R	SRA rd, rt, sa	<u>Shift Word Right Arithmetic</u>: The contents of the low-order word of general register *rt* are shifter right by *sa* bits, sign extending the high order bits. The result is placed in register *rd.*
SRAV	R	SRAV rd, rt, sa	<u>Shift Word Right Arithmetic Variable:</u> The contents of the low-order word of general register *rt* are shifted right by the low-order five bits contained in general register *rs* , sign extending the high-order bits. The word result is placed in register *rd* .
SRL	R	SRL rd, rt, sa	<u>Shift Word Right Logical:</u> The contents of the low-order word of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The word result is placed in register *rd* .
SRLV	R	SRLV rd, rt, rs	<u>Shift Word Right Logical Variable:</u> The contents of the low-order word of general register *rt* are shifted right by the low-order five bits contained in general register *rs* , inserting zeros into the high-order bits. The word result is placed in register *rd* .
SUB	R	SUB rd, rs, rt	<u>Subtract Word</u>: The word value in general register *rt* is subtracted from the word value in general register *rs* and the result is placed into general register *rd* . If the subtraction results in 32-bit 2's complement arithmetic overflow, the overflow flag is set.
SUBU	R	SUBU rd, rs, rt	<u>Subtract Unsigned Word:</u> The word value in general register *rt* is subtracted from the word value in general register *rs* and the result is placed into general register *rd.* If the subtraction results in 32-bit 2's complement arithmetic overflow, the overflow flag is not set.
SW	I	SW rt, offset(base)	<u>Store Word</u>: The 16 bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual addres. The contents of the word in general register *rt* are stored at the memory location specified by the effective address.
XOR	R	OR rd, rs, rt	<u>Exclusive OR</u>: The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical XOR operation. The result is placed into register *rd.*
XORI	I	XORI rt, rs, immediate	<u>Exclusive OR Immediate:</u> The 16 bit immediate is zero-extended and combined with the contents of general register *rs* in a bit-wise logical XOR operation. The result is placed into general register *rt* .

APPENDIX B: SCHEMATICS AND VHDL CODE

Appendix B contains all the schematics and VHDL code for components not previously included

A PIX PROCESSOR

1. IF Phase Combinatorial Logic (pc_cl.sch)



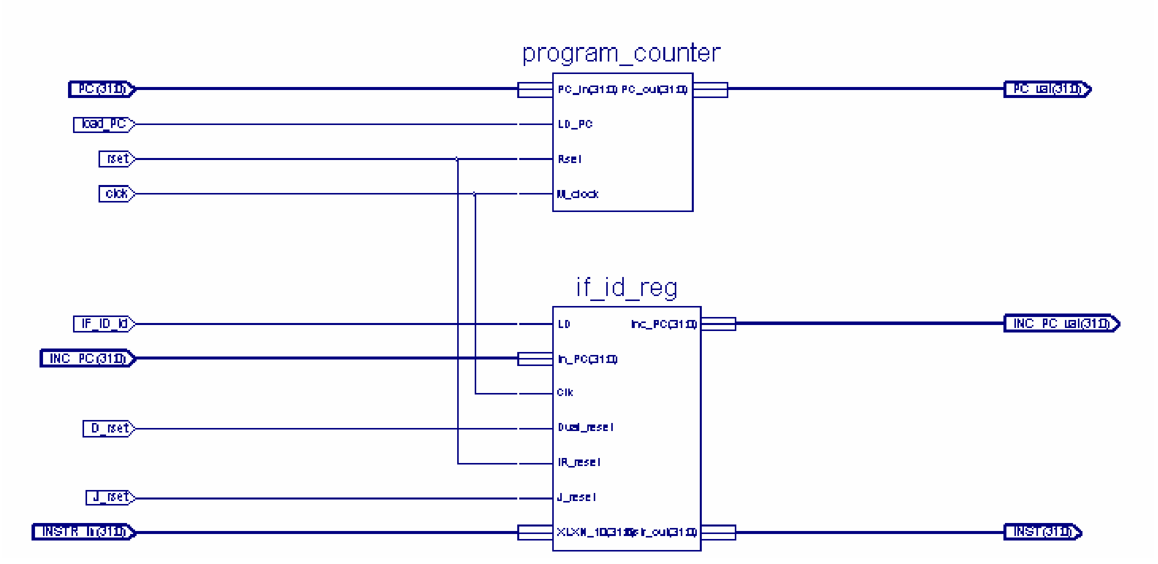
2. Program Counter Incrementor (PC_inc.vhd)

```
entity PC_inc is
   Port ( PC_in : in std_logic_vector(31 downto 0);
          PC_out : out std_logic_vector(31 downto 0));
end PC_inc;

architecture Behavioral of PC_inc is

begin
  process ( PC_in )
    begin
           PC_out <= "00000000000000000000000000000000";
           PC_out <= PC_in + "00000000000000000000000000000001";
    end process;
end Behavioral;
```

3. Program Counter Multiplexer PC_mux.vhd)

```
entity PC_MUX is
   Port ( Inc_PC : in std_logic_vector(31 downto 0);
          Brnch  : in std_logic_vector(31 downto 0);
          Jump   : in std_logic_vector(31 downto 0);
                Intrpt : in std_logic_vector(31 downto 0);
          Sel  : in std_logic_vector(2 downto 0);
          Data_out : out std_logic_vector(31 downto 0));
end PC_MUX;

architecture Behavioral of PC_MUX is

begin
   process (Inc_PC, Brnch, Jump, Intrpt, Sel)
      begin
           Data_out <= "00000000000000000000000000000000";
           if Sel = "000" then Data_out <= Inc_PC;
           elsif Sel = "001" then Data_out <= Brnch;
           elsif Sel = "010" then Data_out <= Jump;
           else  Data_out <= Intrpt;


           end if;
         end process;


end Behavioral;
```

B.. IF/ID REGISTER (PC_IR_REG.SCH)



a. Instruction Register (instr_reg.vhd)

```
entity instr_reg is
   Port ( Instr_in   : in std_logic_vector(31 downto 0);
          Instr_LD   : in std_logic;
          Reset      : in std_logic;
          Rset_jmp   : in std_logic;
          Rset_brnch : in std_logic;
          clk        : in std_logic;
--          I_out31_26 : out std_logic_vector(5 downto 0);
--          I_out25_21 : out std_logic_vector(4 downto 0);
--          I_out25_0  : out std_logic_vector(25 downto 0);
```

```
 --         I_out20_16 : out std_logic_vector(4 downto 0);
 --         I_out15_0  : out std_logic_vector(15 downto 0);
 --         I_out15_11 : out std_logic_vector(4 downto 0);
 --         I_out5_0   : out std_logic_vector(5 downto 0);
 --         I_out10_6  : out std_logic_vector(4 downto 0));
            I_out      : out std_logic_vector(31 downto 0));
end instr_reg;

architecture Behavioral of instr_reg is

begin
   process(Instr_in, Instr_LD, Reset, Rset_jmp, Rset_brnch, clk)
      begin
            if (clk'event AND clk = '1')then
               if (reset = '1' OR Rset_jmp = '1' OR Rset_brnch = '1')
                        then
--                                  I_out31_26  <= "000000";
--                  I_out25_21 <= "00000";
--                  I_out25_0  <= "00000000000000000000000000";
--                           I_out20_16 <= "00000";
--                  I_out15_0  <= "0000000000000000";
--                  I_out15_11 <= "00000";
--                  I_out5_0   <= "000000";
--                  I_out10_6  <= "00000";
                    I_out  <= "00000000000000000000000000000000";
               elsif Instr_LD = '1'
                        then
--                                  I_out31_26 <= Instr_in(31 downto 26);
--                  I_out25_21 <= Instr_in(25 downto 21);
--                  I_out25_0  <= Instr_in(25 downto 0);
--                           I_out20_16 <= Instr_in(20 downto 16);
--                  I_out15_0  <= Instr_in(15 downto 0);
--                  I_out15_11 <= Instr_in(15 downto 11);
--                  I_out5_0   <= Instr_in(5 downto 0);
--                  I_out10_6  <= Instr_in(10 downto 6);
                    I_out  <= Instr_in;
               end if;
            end if;
         end process;


end Behavioral;
```

1. ID Phase Combinatorial Logic

a. Branch Address Adder (br_addr_adder.vhd)

```
entity Br_Addr_Adder is
   Port ( PC_Addr : in std_logic_vector(31 downto 0);
          Br_Offset : in std_logic_vector(31 downto 0);
          Br_Addr : out std_logic_vector(31 downto 0));
end Br_Addr_Adder;

architecture Behavioral of Br_Addr_Adder is

begin
```

```
             process (PC_Addr, Br_Offset)
               begin
                 Br_Addr <= (PC_Addr + Br_Offset);
             end process;
end Behavioral;
```

b. Comparator (comparator.vhd)

```
entity Comparator is
    Port ( In_A : in std_logic_vector(31 downto 0);
           In_B : in std_logic_vector(31 downto 0);
           A_equal_B : out std_logic;
           A_GEZ :     out std_logic;
           A_GTZ :     out std_logic;
           A_LTZ :     out std_logic);
end Comparator;

architecture Behavioral of Comparator is

begin
   process (In_A, In_B)
      begin
               A_equal_B <= '0';
               A_GEZ <= '0';
        A_GTZ <= '0';
        A_LTZ <= '0';
               if In_A = In_B then A_equal_B <= '1';
               end if;
               if In_A(31) = '0' then A_GEZ <= '1';
        end if;
               if ((In_A > "00000000000000000000000000000000") and (In_A(31) =
               '0')) then A_GTZ <= '1';
               end if;
               if In_A(31) = '1' then A_LTZ <= '1';
               end if;
   end process;
end Behavioral;
```

c. Link Controller (link_control.vhd)

```
entity Link_Cntrl is
    Port ( Reg_rd : in std_logic_vector(4 downto 0);
           Link : in std_logic;
           JALR : in std_logic;
           Link_Reg : out std_logic_vector(4 downto 0));
end Link_Cntrl;

architecture Behavioral of Link_Cntrl is

begin
   process (Link, JALR, Reg_rd)
      begin
                Link_Reg <= "00000";
                           if ((Link = '1') AND (JALR = '1') AND (Reg_rd /= "00000")) OR
                          (Link = '0') then
                           Link_Reg <= Reg_rd;
                else
                Link_Reg <= "11111";
              end if;
            end process;
end Behavioral;
```

d. Main Controller (main_control.vhd)

```
entity Main_Control is
   Port ( Opcode        : in std_logic_vector(5 downto 0);
              Funct_Fld      : in std_logic_vector(5 downto 0);
        Brnch_Type     : in std_logic_vector(4 downto 0);
        Ovr_Flw        : in std_logic;
              ALU_Op         : out std_logic_vector(2 downto 0);
              RegDst         : out std_logic;
        ALU_Src        : out std_logic;
        MemRd          : out std_logic;
        MemWrt         : out std_logic;
        RegWrt         : out std_logic;
        MemToReg       : out std_logic;
              BNE                        : out std_logic;
              BEQ                    : out std_logic;
              BLEZ               : out std_logic;
              BGEZ               : out std_logic;
              BGEZAL             : out std_logic;
              BGTZ               : out std_logic;
              BLTZ               : out std_logic;
              BLTZAL             : out std_logic;
              JUMP           : out std_logic;
              JALR           : out std_logic;
              J_R            : out std_logic;
              J_EPC              : out std_logic;
              LINK           : out std_logic;
              Logic_Imm      : out std_logic;
              EX_flush       : out std_logic);

end Main_Control;

architecture Behavioral of Main_Control is

begin
  process (Opcode, Funct_Fld, Brnch_Type, Ovr_Flw)
    begin
       ALU_Op   <= "000";
                  RegDst   <=   '0';
                  ALU_Src  <=   '0';
                  MemRd    <=   '0';
                  MemWrt   <=   '0';
                  RegWrt   <=   '0';
                  MemToReg <=   '0';
                  BNE         <=   '0';
             BEQ    <=   '0';
             BLEZ             <=   '0';
             BGEZ             <=   '0';
             BGEZAL   <=   '0';
             BGTZ             <=   '0';
```

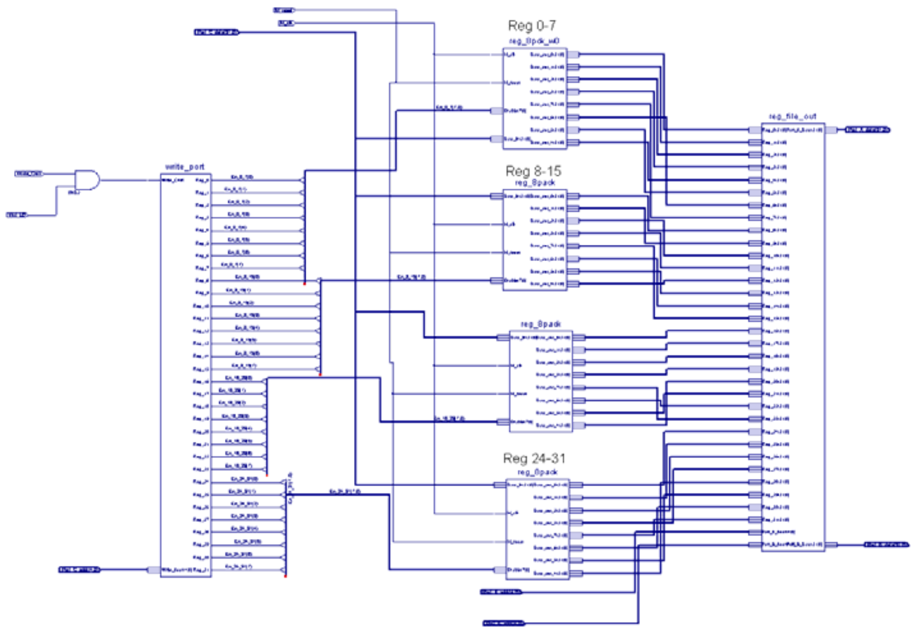
```
BLTZ            <=   '0';
BLTZAL   <=   '0';
JUMP     <=   '0';
        JALR     <=   '0';
        J_R      <=   '0';
        J_EPC    <=   '0';
        LINK     <=   '0';
        Logic_Imm <=  '0';
        EX_flush  <=  '0';

        if (Ovr_Flw = '1') then    --  ALU Op resulted in Overflow
                  EX_flush <= '1';                  --   prevents result from
        being written to "rd"
        end if;

        if (Opcode = "001100" OR Opcode = "001101" OR Opcode =
        "001110") then
           Logic_Imm <= '1';       -- ANDI, ORI, XORI   (needed for sign
        xtndr)
        end if;

        case Opcode is
          when "000000" =>                -- R type instruction
         ALU_Op   <= "010";
                     RegDst   <=   '1';
                     ALU_Src  <=   '0';
                     MemRd    <=   '0';
                     MemWrt   <=   '0';

                     RegWrt   <=   '1';
                     MemToReg <=   '0';
                     case Funct_Fld is
                                when "001000" =>
                       J_R  <= '1';                -- J_R instruction (0/8)
                             JUMP <= '1';
                          when "001001" =>
                       LINK <= '1';                -- JALR instruction  (0/9)
                             JALR <= '1';
                             JUMP <= '1';
                          when "001010" =>
                       J_EPC <= '1';-- J_EPC instruction (0/10)
                             JUMP  <= '1';
                   when others => null;
                     end case;

           when "000001" =>              -- PC relative Branch
         ALU_Op   <= "000";
                     RegDst   <=   '0';
                     ALU_Src  <=   '1';
                     MemRd    <=   '0';
                     MemWrt   <=   '0';
             RegWrt   <=   '1';
                     MemToReg <=   '0';
                     case Brnch_Type is
                        when "00000" =>
                          BLTZ   <= '1';
```

```
                                   when "00001" =>
                                      BGEZ   <= '1';
                                   when "10000" =>
                                      BLTZAL   <= '1';
                                      LINK     <= '1';
                                      RegDst   <= '1';
                                   when "10001" =>
                                      BGEZAL   <= '1';
                                      LINK     <= '1';
                                      RegDst   <= '1';
                                         when others => null;
                                end case;
                          when "000010" =>              -- J (jump) (2)
                         ALU_Op   <= "000";
                                RegDst   <=   '0';
                                ALU_Src  <=   '0';
                                MemRd    <=   '0';
                                MemWrt   <=   '0';
                                RegWrt   <=   '0';
                                MemToReg <=   '0';
                                JUMP     <=   '1';
                  when "000011" =>              -- JAL (jump & link) (3)
                         ALU_Op   <= "000";
                                RegDst   <=   '1';
                                ALU_Src  <=   '0';
                                MemRd    <=   '0';
                                MemWrt   <=   '0';
                                RegWrt   <=   '1';
                                MemToReg <=   '0';
                                JUMP     <=   '1';
                                LINK     <=   '1';
                          when "000100"  =>              -- BEQ  (4)
                         ALU_Op   <= "000";
                                RegDst   <=   '0';
                                ALU_Src  <=   '0';
                                MemRd    <=   '0';
                                MemWrt   <=   '0';
                                RegWrt   <=   '0';
                                MemToReg <=   '0';
                                BEQ      <=   '1';
                  when "000101" =>               -- BNE  (5)
                         ALU_Op   <= "000";
                                RegDst   <=   '0';
                                ALU_Src  <=   '0';
                                MemRd    <=   '0';
                                MemWrt   <=   '0';
                                RegWrt   <=   '0';
                                MemToReg <=   '0';
                                BNE      <=   '1';
                          when "000110" =>               -- BLEZ (6)
                         ALU_Op   <= "000";
                                RegDst   <=   '0';
                                ALU_Src  <=   '0';
                                MemRd    <=   '0';
                                MemWrt   <=   '0';
                                RegWrt   <=   '0';
```

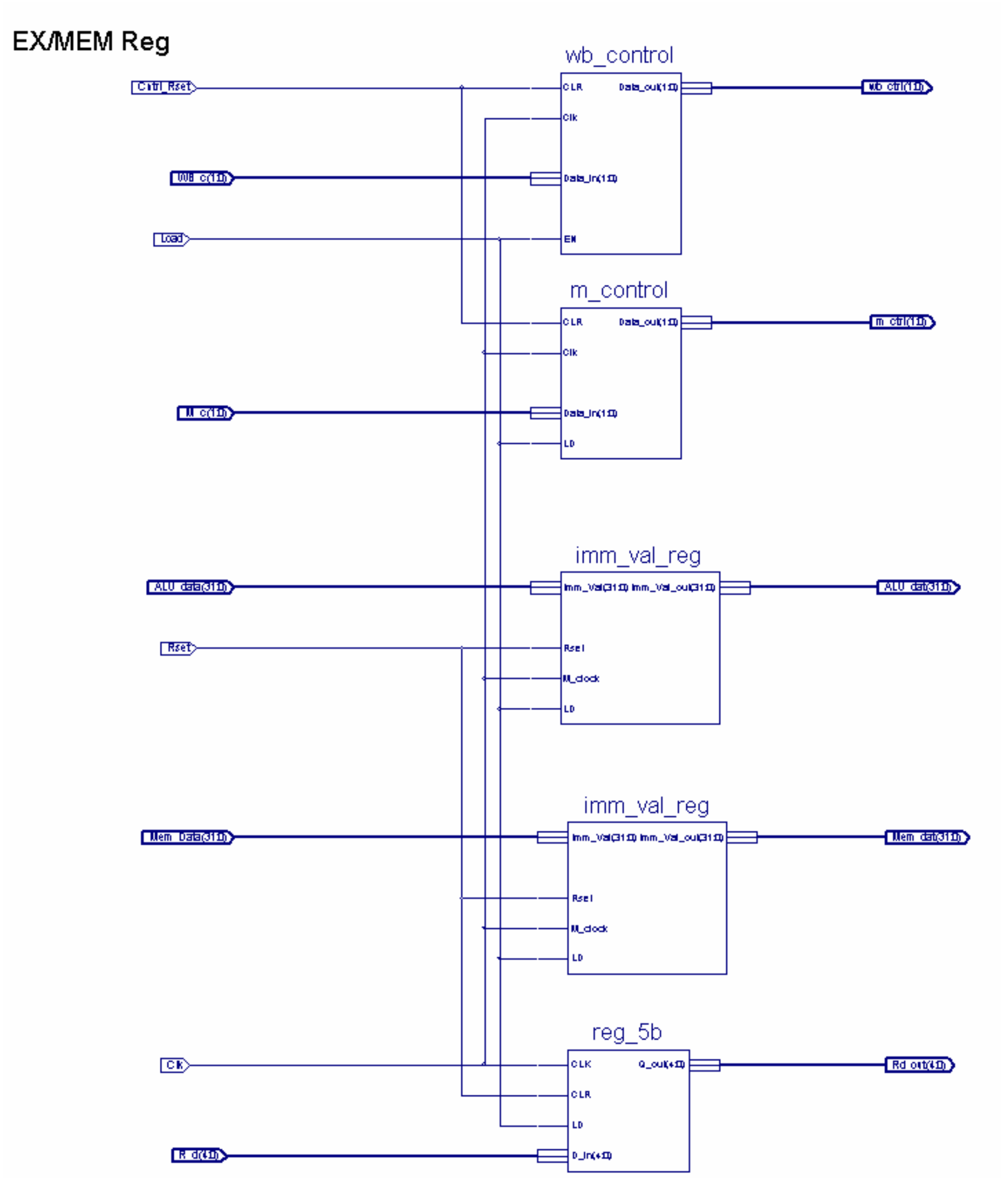
```
                MemToReg <=   '0';
                BLEZ     <=   '1';
         when "000111" =>               -- BGTZ (7)
       ALU_Op   <= "000";
                RegDst   <=   '0';
                ALU_Src  <=   '0';
                MemRd    <=   '0';
                MemWrt   <=   '0';
                RegWrt   <=   '0';
                MemToReg <=   '0';
                BGTZ     <=   '1';
         when "001000" =>               -- ADDI (8)
       ALU_Op   <= "011";
                RegDst   <=   '0';
                ALU_Src  <=   '1';
                MemRd    <=   '0';
                MemWrt   <=   '0';
                RegWrt   <=   '1';
                MemToReg <=   '0';
         when "001001" =>               -- ADDIU (9)
       ALU_Op   <= "000";
                RegDst   <=   '0';
                ALU_Src  <=   '1';
                MemRd    <=   '0';
                MemWrt   <=   '0';
                RegWrt   <=   '1';
                MemToReg <=   '0';
         when "001010" =>               -- SLTI (10)
       ALU_Op   <= "111";
                RegDst   <=   '1';
                ALU_Src  <=   '1';
                MemRd    <=   '0';
                MemWrt   <=   '0';
                RegWrt   <=   '1';
                MemToReg <=   '0';
        when "001011" =>                -- SLTIU (11)
     ALU_Op   <= "111";
                RegDst   <=   '1';
                ALU_Src  <=   '1';
                MemRd    <=   '0';
                MemWrt   <=   '0';
                RegWrt   <=   '1';
                MemToReg <=   '0';
        when "001100" =>                -- ANDI (12)
     ALU_Op   <= "100";
                RegDst   <=   '0';
                ALU_Src  <=   '1';
                MemRd    <=   '0';
                MemWrt   <=   '0';
                RegWrt   <=   '1';
                MemToReg <=   '0';
        when "001101" =>               -- ORI (13)
       ALU_Op   <= "101";
                RegDst   <=   '0';
                ALU_Src  <=   '1';
                MemRd    <=   '0';
```

```
                                MemWrt   <=   '0';
                                RegWrt   <=   '1';
                                MemToReg <=   '0';
                    when "001110" =>                -- XORI (14)
                ALU_Op   <= "110";
                                RegDst   <=   '0';
                                ALU_Src  <=   '1';
                                MemRd    <=   '0';
                                MemWrt   <=   '0';
                                RegWrt   <=   '1';
                                MemToReg <=   '0';
                    when "100011" =>                -- LW (35)
                   ALU_Op   <= "000";
                                RegDst   <=   '0';
                                ALU_Src  <=   '1';
                                MemRd    <=   '1';
                                MemWrt   <=   '0';
                                RegWrt   <=   '1';
                                MemToReg <=   '1';
                    when "101011" =>                -- SW (43)
                   ALU_Op   <= "000";
                                RegDst   <=   '0';
                                ALU_Src  <=   '1';
                                MemRd    <=   '0';
                                MemWrt   <=   '1';
                                RegWrt   <=   '0';
                                MemToReg <=   '0';

                    when others => null;
                 end case;
        end process;
end Behavioral;
```

### *e.	Register File (regiser_file.sch)*



2. EX/MEM Phase Register

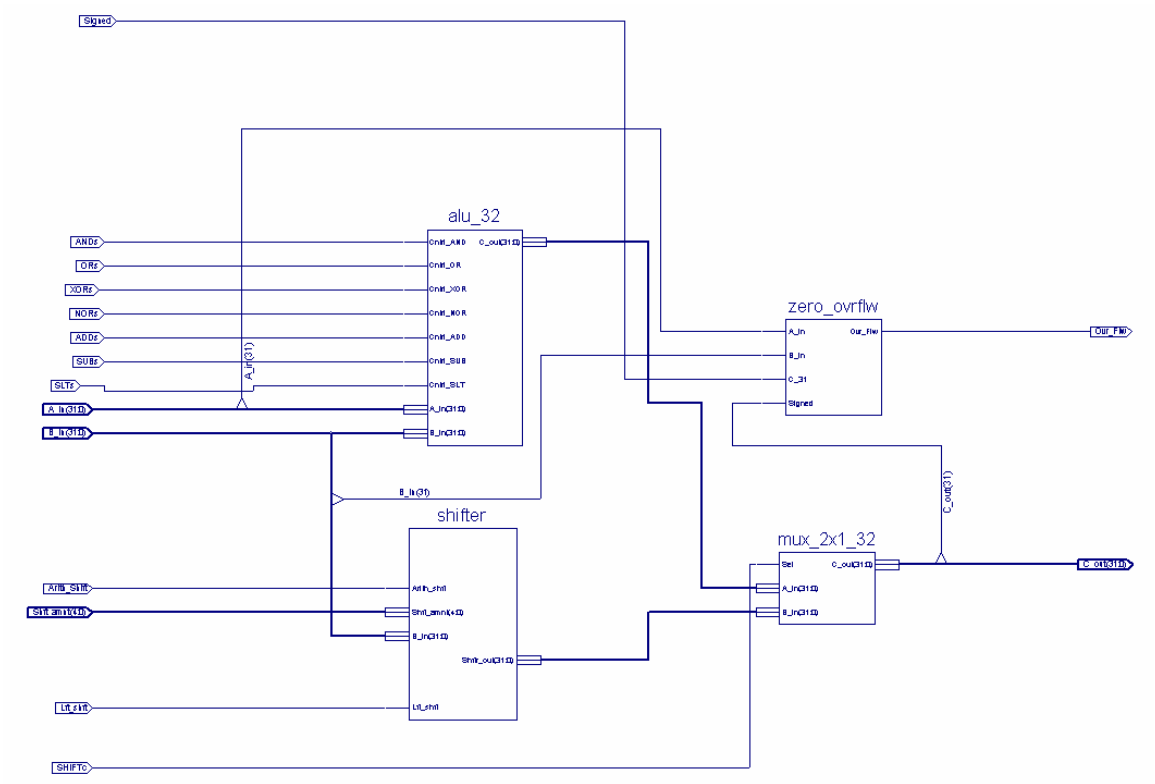


3. EX Phase Combinatorial Logic

a. ALU Control (alu_cntrl.vhd)

```
-- ALU controller

-- Rev date 17May05



-- Function: Module deodes the ALU Op field of the instruction. Depending
--          on the contents, the proper operation signal is asserted. If
--        the field contents is "010", the Function feild is decoded
--        to determine proper operation signal to ALU.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALU_cntrl is

  Port (  Funct_Fld  : in std_logic_vector(5 downto 0);
                  ALU_Op     : in std_logic_vector(2 downto 0);
        ADD        : out std_logic;
        ANDc       : out std_logic;
        ORc        : out std_logic;
        XORc       : out std_logic;
        NORc       : out std_logic;
        SLT        : out std_logic;
             SUB        : out std_logic;
                  Arith_Shft : out std_logic;
                  Shift      : out std_logic;
                  Lft_Shft   : out std_logic;
                  Signed     : out std_logic);

end ALU_cntrl;

architecture Behavioral of ALU_cntrl is

begin
  process(Funct_Fld, ALU_Op)
     begin
            ADD        <= '0';
                 ANDc       <= '0';
                 ORc        <= '0';
                 XORc       <= '0';
                 NORc       <= '0';
                 SLT        <= '0';
                 SUB        <= '0';
                 Arith_Shft <= '0';
                 Shift      <= '0';
                 Lft_Shft   <= '0';
                 Signed     <= '0';

                 case ALU_Op is
                          when "000" => ADD  <= '1';                    --   ADD
                 for LW, SW, ADDI or ADDIU instr
                   when "001" => SUB  <= '1';             -- SUB for Branch
             when "011" => ADD  <= '1';
                         Signed  <= '1';       -- allow Over Flow
                   when "100" => ANDc <= '1';             -- ANDI instr
                   when "101" => ORc  <= '1';             -- ORI instr
                   when "110" => XORc <= '1';             -- XORI instr
                   when "111" => SLT <= '1';              -- SLTI or SLTIU instr
                          when "010" =>                                --  R
                 type instr =>
            case Funct_Fld is                                        --
          look at Function Field
                  when  "100000" => ADD   <= '1';                  -- 32
                              Signed <= '1';       -- allow Over Flow
                      when  "100001" => ADD   <= '1';
          -- 33 ADDU
                  when  "100010" => SUB   <= '1';                  -- 34
                                    Signed <= '1';                 --  allow
          Over Flow
               when  "100011" => SUB   <= '1';              -- 35 SUBU
     when  "100100" => ANDc  <= '1';       -- 36
               when  "100101" => ORc   <= '1';              -- 37
               when  "100110" => XORc  <= '1';              -- 38
               when  "100111" => NORc  <= '1';              -- 39
               when  "101011" => SLT   <= '1';              -- 42
                  when  "101010" => SLT   <= '1';                  --     43
          SLTU
               when  "000000" => Lft_Shft    <= '1';        -- 00
                                 Shift       <= '1';
               when  "000010" => Shift       <= '1';        -- 02
               when  "000011" => Arith_Shft  <= '1';               -- 03
                                 Shift       <= '1';
               when  "000100" => Lft_Shft    <= '1';        -- 04
                                 Shift       <= '1';
               when  "000110" => Shift       <= '1';        -- 06
               when  "000111" => Arith_Shft  <= '1';               -- 07
                                 Shift       <= '1';
                  when others    => null;
                 end case;
           when others    => null;
           end case;
  end process;


end Behavioral;
```

b.　　ALU Shell (super_alu.vhd)

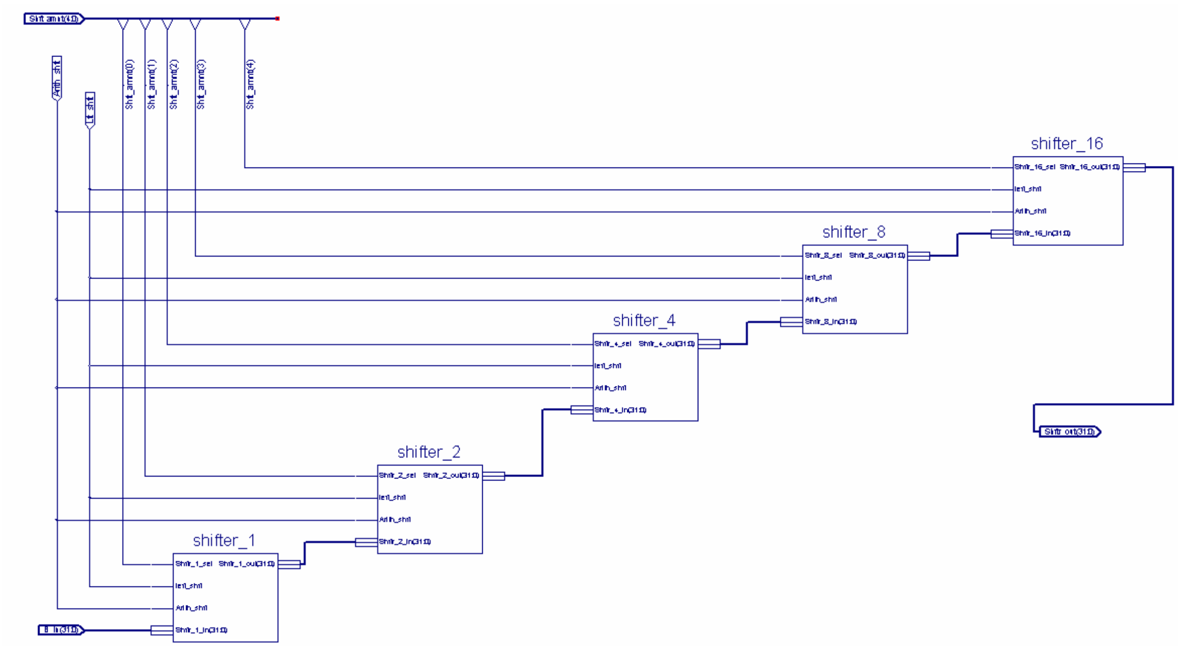


c. ALU-32 bit (alu_32.vhd)

```
entity ALU_32 is
    Port ( A_in        : in std_logic_vector(31 downto 0);
           B_in        : in std_logic_vector(31 downto 0);
           Cntrl_AND   : in std_logic;
           Cntrl_OR    : in std_logic;
                       Cntrl_XOR   : in std_logic;
                       Cntrl_NOR   : in std_logic;
           Cntrl_ADD   : in std_logic;
           Cntrl_SUB   : in std_logic;
                       Cntrl_SLT   : in std_logic;
           C_out       : out std_logic_vector(31 downto 0));
end ALU_32;

architecture Behavioral of ALU_32 is

begin
            process (A_in, B_in, Cntrl_AND, Cntrl_OR, Cntrl_XOR, Cntrl_NOR,
                   Cntrl_ADD, Cntrl_SUB, Cntrl_SLT)
        begin
              C_out <= "00000000000000000000000000000000" ;
              if    (Cntrl_AND = '1') then C_out <= (A_in and B_in);
              elsif (Cntrl_OR = '1') then C_out <= (A_in or B_in);
              elsif (Cntrl_XOR = '1') then C_out <= (A_in xor B_in);
              elsif (Cntrl_NOR = '1') then C_out <= (A_in nor B_in);
              elsif (Cntrl_ADD = '1') then C_out <= (A_in + B_in);
              elsif (Cntrl_SUB = '1') then C_out <= (A_in - B_in);
              elsif (Cntrl_SLT = '1' and A_in < B_in) then
                             C_out <= "00000000000000000000000000000001" ;
        elsif (Cntrl_SLT = '1' and A_in > B_in) then
```

```
          C_out <= "00000000000000000000000000000000" ;
               elsif (Cntrl_SLT = '1' and A_in = B_in) then
                C_out <= "00000000000000000000000000000000" ;
               end if;
             end process;

end Behavioral;
```

d. *Shifter (Shifter.sch)*



e. *Shift_8bit(Shift_8.vhd)*

```
entity shifter_8 is
    Port ( Shftr_8_in  : in std_logic_vector(31 downto 0);
           Shftr_8_sel : in std_logic;
                       left_shft   : in std_logic;
           Arith_shft  : in std_logic;
           Shftr_8_out : out std_logic_vector(31 downto 0));

end shifter_8;

architecture Behavioral of shifter_8 is

begin
   process(Shftr_8_in, Shftr_8_sel, left_shft, Arith_Shft)
      begin
              if Shftr_8_sel = '1' then
                if left_shft = '1' then                    -- Shift Left Logical
                       Shftr_8_out(31 downto 8) <= Shftr_8_in(23 downto 0);
                             Shftr_8_out(7 downto 0) <= "00000000";
                     elsif Arith_shft = '1' then           -- Shift Right Arithmatic
                       Shftr_8_out(23 downto 0) <= Shftr_8_in(31 downto 8);
                         if Shftr_8_in(31) = '1' then
```

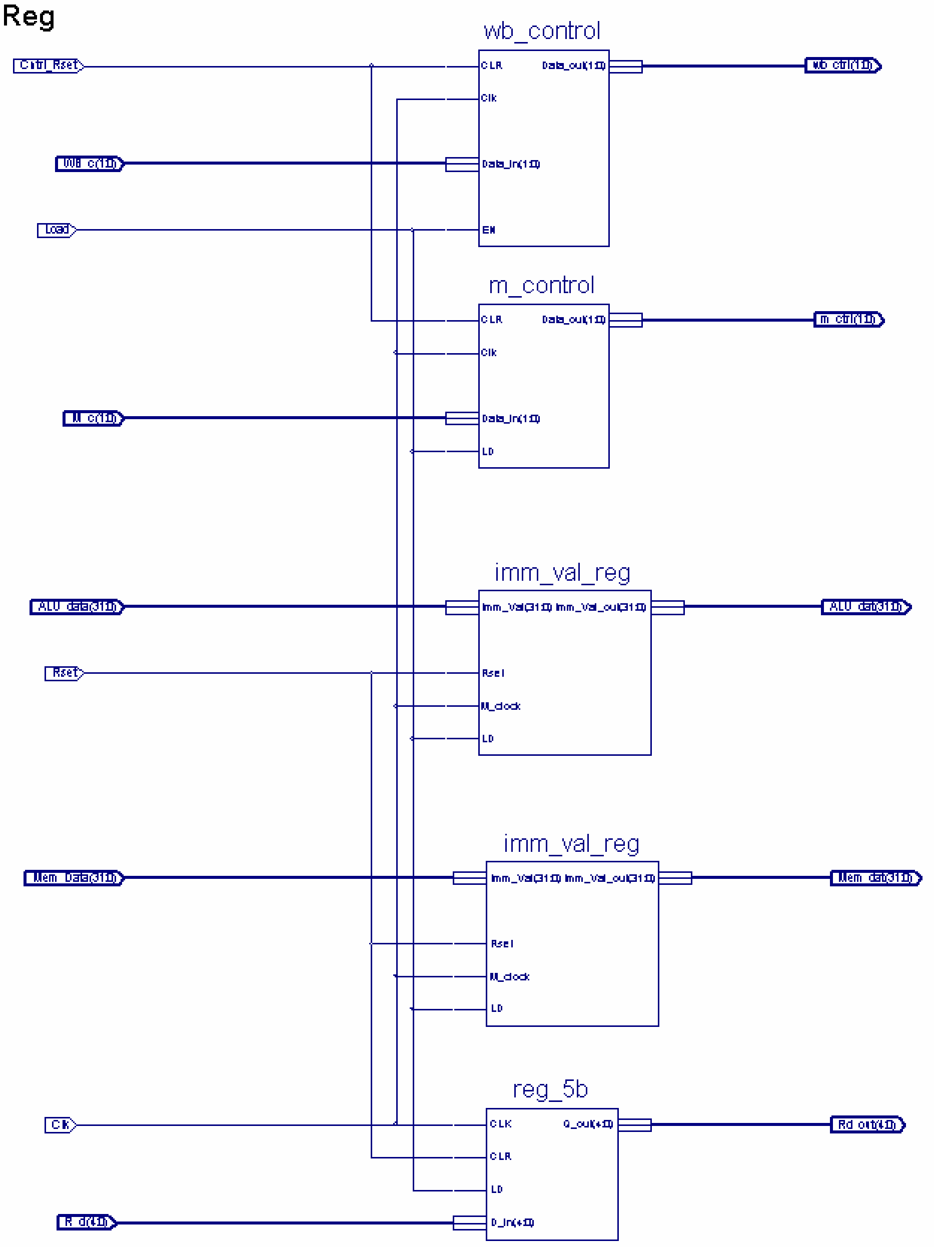
```
                              Shftr_8_out(31 downto 24)      <= "11111111";   -- sra for
        negative value
                              else Shftr_8_out(31 downto 24) <= "00000000";    -- sra
        for positive value
                                end if;
                              else Shftr_8_out(23 downto 0) <= Shftr_8_in(31 downto
        8); -- Shift Right Logical
                          Shftr_8_out(31 downto 24) <= "00000000";
                       end if;
                     else Shftr_8_out <= Shftr_8_in;
                     end if;
  end process;

end Behavioral;
```

f. *Overflow Circuit(zero_ovrflw.vhd.vhd)*

```
entity Zero_OvrFlw is
    Port ( A_in    : in std_logic;
           B_in    : in std_logic;
           C_in    : in std_logic_vector (31 downto 0 );
                   Signed  : in std_logic;
           Ovr_Flw : out std_logic);
end Zero_OvrFlw;

architecture Behavioral of Zero_OvrFlw is

begin
  process(A_in, B_in, C_in, Signed)
     begin
        Ovr_Flw <= '0';
        if (Signed = '1') then
           if ((A_in = '1') and (B_in = '1') and (C_in(31) = '0')) then
              Ovr_Flw <= '1';
           elsif ((A_in = '0') and (B_in = '0') and (C_in(31) = '1')) then
              Ovr_Flw <= '1';
        end if;
     end if;
  end process;
end Behavioral;
```

4. EX Phase Combinatorial Logic

EX/MEM Reg



wb_control
Ctrl_Rset
CLR
Data_out(1:0)
wb ctrl(1:0)
Clk
WB c(1:0)
Data_in(1:0)
Load
EN
m_control
CLR
Data_out(1:0)
m ctrl(1:0)
Clk
M c(1:0)
Data_in(1:0)
LD
imm_val_reg
ALU data(31:0)
Imm_Val(31:0)
Imm_Val_out(31:0)
ALU dat(31:0)
Rset
Rsel
M_clock
LD
imm_val_reg
Mem Data(31:0)
Imm_Val(31:0)
Imm_Val_out(31:0)
Mem dat(31:0)
Rsel
M_clock
LD
reg_5b
Clk
CLK
Q_out(4:0)
Rd out(4:0)
CLR
LD
R d(4:0)
D_in(4:0)

APPENDIX C: CODE FOR DECODING ERROR SYNDROME

```
Variables (either integer or Boolean)

PhaseIF
PhaseEX
PhaseMEM
PhaseWB

ProcA
ProcB
ProcC

Begin

if ES(28) = '1' then ProcA        = '1';
if ES(27) = '1' then ProcB        = '1';
if ES(26) = '1' then ProcC        = '1';

if ES(25) = '1' then PhaseWB    = '1';
if ES(24) = '1' then PhaseMEM = '1';
if ES(23) = '1' then PhaseEX     = '1';
if ES(22) = '1' then PhaseIF       = '1';

** Break down for WB Phase voters
if ES(25) AND ES(10) = '1' then ** voter 1
                                begin
                                        if ES(0) = '1' then println("Mem_Data(0)");
                                        if ES(1) = '1' then println("Mem_Data(1)");
                                        if ES(2) = '1' then println("Mem_Data(2)");
                                        if ES(3) = '1' then println("Mem_Data(3)");
                                        if ES(4) = '1' then println("Mem_Data(4)");
                                        if ES(5) = '1' then println("Mem_Data(5)");
                                        if ES(6) = '1' then println("Mem_Data(6)");
                                        if ES(7) = '1' then println("Mem_Data(7)");
                                        if ES(8) = '1' then println("Mem_Data(8)");
                                end;

 if ES(25) AND ES(11) = '1' then ** voter 2
                                begin
                                        if ES(0) = '1' then println("Mem_Data(9)");
                                        if ES(1) = '1' then println("Mem_Data(10)");
                                        if ES(2) = '1' then println("Mem_Data(11)");
                                        if ES(3) = '1' then println("Mem_Data(12)");
                                        if ES(4) = '1' then println("Mem_Data(13)");
                                        if ES(5) = '1' then println("Mem_Data(14)");
                                        if ES(6) = '1' then println("Mem_Data(15)");
                                        if ES(7) = '1' then println("Mem_Data(16)");
                                        if ES(8) = '1' then println("Mem_Data(17)");
                                end;

if ES(25) AND ES(12) = '1' then ** voter 3
                                begin
                                        if ES(0) = '1' then println("Mem_Data(18)");
```

```
                                                if ES(1) = '1' then println("Mem_Data(19)");
                                                if ES(2) = '1' then println("Mem_Data(20)");
                                                if ES(3) = '1' then println("Mem_Data(21)");
                                                if ES(4) = '1' then println("Mem_Data(22)");
                                                if ES(5) = '1' then println("Mem_Data(23)");
                                                if ES(6) = '1' then println("Mem_Data(24)");
                                                if ES(7) = '1' then println("Mem_Data(25)");
                                                if ES(8) = '1' then println("Mem_Data(26)");
                                        end;

if ES(25) AND ES(13) = '1' then         ** voter 4
                                        begin
                                                if ES(0) = '1' then println("Mem_Data(27)");
                                                if ES(1) = '1' then println("Mem_Data(28)");
                                                if ES(2) = '1' then println("Mem_Data(29)");
                                                if ES(3) = '1' then println("Mem_Data(30)");
                                                if ES(4) = '1' then println("Mem_Data(31)");
                                                if ES(5) = '1' then println("Reg_Data(0)");
                                                if ES(6) = '1' then println("Reg_Data(1)");
                                                if ES(7) = '1' then println("Reg_Data(2)");
                                                if ES(8) = '1' then println("Reg_Data(3)");
                                        end;

if ES(25) AND ES(14) = '1' then         ** voter 5
                                        begin
                                                if ES(0) = '1' then println("Reg_Data(4)");
                                                if ES(1) = '1' then println("Reg_Data(5)");
                                                if ES(2) = '1' then println("Reg_Data(6)");
                                                if ES(3) = '1' then println("Reg_Data(7)");
                                                if ES(4) = '1' then println("Reg_Data(8)");
                                                if ES(5) = '1' then println("Reg_Data(9)");
                                                if ES(6) = '1' then println("Reg_Data(10)");
                                                if ES(7) = '1' then println("Reg_Data(11)");
                                                if ES(8) = '1' then println("Reg_Data(12)");
                                        end;

if ES(25) AND ES(15) = '1' then         ** voter 6
                                        begin
                                                if ES(0) = '1' then println("Reg_Data(13)");
                                                if ES(1) = '1' then println("Reg_Data(14)");
                                                if ES(2) = '1' then println("Reg_Data(15)");
                                                if ES(3) = '1' then println("Reg_Data(16)");
                                                if ES(4) = '1' then println("Reg_Data(17)");
                                                if ES(5) = '1' then println("Reg_Data(18)");
                                                if ES(6) = '1' then println("Reg_Data(19)");
                                                if ES(7) = '1' then println("Reg_Data(20)");
                                                if ES(8) = '1' then println("Reg_Data(21)");
                                        end;

if ES(25) AND ES(16) = '1' then         ** voter 7
                                        begin
                                                if ES(0) = '1' then println("Reg_Data(22)");
                                                if ES(1) = '1' then println("Reg_Data(23)");
                                                if ES(2) = '1' then println("Reg_Data(24)");
                                                if ES(3) = '1' then println("Reg_Data(25)");
                                                if ES(4) = '1' then println("Reg_Data(26)");
                                        if ES(5) = '1' then println("Reg_Data(27)");
                                        if ES(6) = '1' then println("Reg_Data(28)");
                                        if ES(7) = '1' then println("Reg_Data(29)");
                                        if ES(8) = '1' then println("Reg_Data(30)");
                                end;

if ES(25) AND ES(17) = '1' then ** voter 8
                                begin
                                        if ES(0) = '1' then println("Reg_Data(31)");
                                        if ES(1) = '1' then println("Reg_Sel(0)");
                                        if ES(2) = '1' then println("Reg_Sel(1)");
                                        if ES(3) = '1' then println("Reg_Sel(2)");
                                        if ES(4) = '1' then println("Reg_Sel(3)");
                                        if ES(5) = '1' then println("Reg_Sel(4)");
                                        if ES(6) = '1' then println("WB_cnrl(0)");
                                        if ES(7) = '1' then println("WB_cntrl(1)");
                                end;

** Break down for MEM Phase voters
if ES(24) AND ES(10) = '1' then ** voter 1
                                begin
                                        if ES(0) = '1' then println("Rd_out(0)");
                                        if ES(1) = '1' then println("Rd_out(1)");
                                        if ES(2) = '1' then println("Rd_out(2)");
                                        if ES(3) = '1' then println("Rd_out(3)");
                                        if ES(4) = '1' then println("Rd_out(4)");
                                        if ES(5) = '1' then println("Mem_Data(0)");
                                        if ES(6) = '1' then println("Mem_Data(1)");
                                        if ES(7) = '1' then println("Mem_Data(2)");
                                        if ES(8) = '1' then println("Mem_Data(3)");
                                        if ES(9) = '1' then println("Mem_Data(4)");
                                end;

 if ES(24) AND ES(11) = '1' then ** voter 2
                                begin
                                        if ES(0) = '1' then println("Mem_Data(5)");
                                        if ES(1) = '1' then println("Mem_Data(6)");
                                        if ES(2) = '1' then println("Mem_Data(7)");
                                        if ES(3) = '1' then println("Mem_Data(8)");
                                        if ES(4) = '1' then println("Mem_Data(9)");
                                        if ES(5) = '1' then println("Mem_Data(10)");
                                        if ES(6) = '1' then println("Mem_Data(11)");
                                        if ES(7) = '1' then println("Mem_Data(12)");
                                        if ES(8) = '1' then println("Mem_Data(13)");
                                        if ES(9) = '1' then println("Mem_Data(14)");
                                end;

if ES(24) AND ES(12) = '1' then ** voter 3
                                begin
                                        if ES(0) = '1' then println("Mem_Data(15)");
                                        if ES(1) = '1' then println("Mem_Data(16)");
                                        if ES(2) = '1' then println("Mem_Data(17)");
                                        if ES(3) = '1' then println("Mem_Data(18)");
                                        if ES(4) = '1' then println("Mem_Data(19)");
                                        if ES(5) = '1' then println("Mem_Data(20)");
                                        if ES(6) = '1' then println("Mem_Data(21)");
```

```
                                                if ES(7) = '1' then println("Mem_Data(22)");
                                                if ES(8) = '1' then println("Mem_Data(23)");
                                                if ES(9) = '1' then println("Mem_Data(24)");
                                        end;
if ES(24) AND ES(13) = '1' then         ** voter 4
                                        begin
                                                if ES(0) = '1' then println("Mem_Data(25)");
                                                if ES(1) = '1' then println("Mem_Data(26)");
                                                if ES(2) = '1' then println("Mem_Data(27)");
                                                if ES(3) = '1' then println("Mem_Data(28)");
                                                if ES(4) = '1' then println("Mem_Data(29)");
                                                if ES(5) = '1' then println("Mem_Data(30)");
                                                if ES(6) = '1' then println("Mem_Data(31)");
                                                if ES(7) = '1' then println("ALU_Data(0)");
                                                if ES(8) = '1' then println("ALU_Data(1)");
                                                if ES(9) = '1' then println("ALU_Data(2)");
                                        end
if ES(24) AND ES(14) = '1' then         ** voter 5
                                        begin
                                                if ES(0) = '1' then println("ALU_Data(3)");
                                                if ES(1) = '1' then println("ALU_Data(4)");
                                                if ES(2) = '1' then println("ALU_Data(5)");
                                                if ES(3) = '1' then println("ALU_Data(6)");
                                                if ES(4) = '1' then println("ALU_Data(7)");
                                                if ES(5) = '1' then println("ALU_Data(8)");
                                                if ES(6) = '1' then println("ALU_Data(9)");
                                                if ES(7) = '1' then println("ALU_Data(10)");
                                                if ES(8) = '1' then println("ALU_Data(11)");
                                                if ES(9) = '1' then println("ALU_Data(12)");
                                        end;

if ES(24) AND ES(15) = '1' then         ** voter 6
                                        begin
                                                if ES(0) = '1' then println("ALU_Data(13)");
                                                if ES(1) = '1' then println("ALU_Data(14)");
                                                if ES(2) = '1' then println("ALU_Data(15)");
                                                if ES(3) = '1' then println("ALU_Data(16)");
                                                if ES(4) = '1' then println("ALU_Data(17)");
                                                if ES(5) = '1' then println("ALU_Data(18)");
                                                if ES(6) = '1' then println("ALU_Data(19)");
                                                if ES(7) = '1' then println("ALU_Data(20)");
                                                if ES(8) = '1' then println("ALU_Data(21)");
                                                if ES(9) = '1' then println("ALU_Data(22)");
                                        end;

if ES(24) AND ES(16) = '1' then         ** voter 7
                                        begin
                                                if ES(0) = '1' then println("ALU_Data(23)");
                                                if ES(1) = '1' then println("ALU_Data(24)");
                                                if ES(2) = '1' then println("ALU_Data(25)");
                                                if ES(3) = '1' then println("ALU_Data(26)");
                                                if ES(4) = '1' then println("ALU_Data(27)");
                                                if ES(5) = '1' then println("ALU_Data(28)");
                                                if ES(6) = '1' then println("ALU_Data(29)");
                                                if ES(7) = '1' then println("ALU_Data(30)");
                                                if ES(8) = '1' then println("ALU_Data(31)");
```

```
                                  end;

if ES(24) AND ES(17) = '1' then  ** voter 8
                                  begin
                                           if ES(0) = '1' then println("M_cntrl(31)");
                                           if ES(1) = '1' then println("M_cntrl(0)");
                                           if ES(2) = '1' then println("WB_cnrl(0)");
                                           if ES(3) = '1' then println("WB_cntrl(1)");
                                  end;

** Break down for EX Phase voters
if ES(23) AND ES(10) = '1' then  ** voter 1
                                  begin
                                           if ES(0) = '1' then println("ID_EX_cntrl(0)");
                                           if ES(1) = '1' then println("ID_EX_cntrl(1)");
                                           if ES(2) = '1' then println("ID_EX_cntrl(2)");
                                           if ES(3) = '1' then println("ID_EX_cntrl(3)");
                                           if ES(4) = '1' then println("ID_EX_cntrl(4)");
                                           if ES(5) = '1' then println("ID_EX_cntrl(5)");
                                           if ES(6) = '1' then println("ID_EX_cntrl(6)");
                                           if ES(7) = '1' then println("ID_EX_cntrl(7)");
                                           if ES(8) = '1' then println("ID_EX_cntrl(8)");
                                           if ES(9) = '1' then println("ALU_A_1(0)");
                                  end;
 if ES(23) AND ES(11) = '1' then ** voter 2
                                  begin
                                           if ES(0) = '1' then println("ALU_A_1(1)");
                                           if ES(1) = '1' then println("ALU_A_1(2)");
                                           if ES(2) = '1' then println("ALU_A_1(3)");
                                           if ES(3) = '1' then println("ALU_A_1(4)");
                                           if ES(4) = '1' then println("ALU_A_1(5)");
                                           if ES(5) = '1' then println("ALU_A_1(6)");
                                           if ES(6) = '1' then println("ALU_A_1(7)");
                                           if ES(7) = '1' then println("ALU_A_1(8)");
                                           if ES(8) = '1' then println("ALU_A_1(9)");
                                           if ES(9) = '1' then println("ALU_A_1(10)");
                                  end;

if ES(23) AND ES(12) = '1' then  ** voter 3
                                  begin
                                           if ES(0) = '1' then println("ALU_A(11)");
                                           if ES(1) = '1' then println("ALU_A(12)");
                                           if ES(2) = '1' then println("ALU_A(13)");
                                           if ES(3) = '1' then println("ALU_A(14)");
                                           if ES(4) = '1' then println("ALU_A(15)");
                                           if ES(5) = '1' then println("ALU_A(16)");
                                           if ES(6) = '1' then println("ALU_A(17)");
                                           if ES(7) = '1' then println("ALU_A(18)");
                                           if ES(8) = '1' then println("ALU_A(19)");
                                           if ES(9) = '1' then println("ALU_A(20)");
                                  end;

if ES(23) AND ES(13) = '1' then  ** voter 4
                                  begin
                                           if ES(0) = '1' then println("ALU_A(21)");
                                           if ES(1) = '1' then println("ALU_A(22)");
```

```
                                                if ES(2) = '1' then println("ALU_A(23)");
                                                if ES(3) = '1' then println("ALU_A(24)");
                                                if ES(4) = '1' then println("ALU_A(25)");
                                                if ES(5) = '1' then println("ALU_A(26)");
                                                if ES(6) = '1' then println("ALU_A(27)");
                                                if ES(7) = '1' then println("ALU_A(28)");
                                                if ES(8) = '1' then println("ALU_A(29)");
                                                if ES(9) = '1' then println("ALU_A(30)");
                                        end;

if ES(23) AND ES(14) = '1' then         ** voter 5
                                        begin
                                                if ES(0) = '1' then println("ALU_A(31)");
                                                if ES(1) = '1' then println("ALU_B(0)");
                                                if ES(2) = '1' then println("ALU_B(1)");
                                                if ES(3) = '1' then println("ALU_B(2)");
                                                if ES(4) = '1' then println("ALU_B(3)");
                                                if ES(5) = '1' then println("ALU_B(4)");
                                                if ES(6) = '1' then println("ALU_B(5)");
                                                if ES(7) = '1' then println("ALU_B(6)");
                                                if ES(8) = '1' then println("ALU_B(7)");
                                                if ES(9) = '1' then println("ALU_B(8)");
                                        end;

if ES(23) AND ES(15) = '1' then         ** voter 6
                                        begin
                                                if ES(0) = '1' then println("ALU_B(9)");
                                                if ES(1) = '1' then println("ALU_B(10)");
                                                if ES(2) = '1' then println("ALU_B(11)");
                                                if ES(3) = '1' then println("ALU_B(12)");
                                                if ES(4) = '1' then println("ALU_B(13)");
                                                if ES(5) = '1' then println("ALU_B(14)");
                                                if ES(6) = '1' then println("ALU_B(15)");
                                                if ES(7) = '1' then println("ALU_B(16)");
                                                if ES(8) = '1' then println("ALU_B(17)");
                                                if ES(9) = '1' then println("ALU_B(18)");
                                        end;

if ES(23) AND ES(16) = '1' then         ** voter 7
                                        begin
                                                if ES(0) = '1' then println("ALU_B(19)");
                                                if ES(1) = '1' then println("ALU_B(20)");
                                                if ES(2) = '1' then println("ALU_B(21)");
                                                if ES(3) = '1' then println("ALU_B(22)");
                                                if ES(4) = '1' then println("ALU_B(23)");
                                                if ES(5) = '1' then println("ALU_B(24)");
                                                if ES(6) = '1' then println("ALU_B(25)");
                                                if ES(7) = '1' then println("ALU_B(26)");
                                                if ES(8) = '1' then println("ALU_B(27)");
                                                if ES(9) = '1' then println("ALU_B(28)");
                                        end;

if ES(23) AND ES(17) = '1' then         ** voter 8
                                        begin
                                                if ES(0) = '1' then println("ALU_B(29)");
                                                if ES(1) = '1' then println("ALU_B(30)");
                                                if ES(2) = '1' then println("ALU_B(31)");
                                                if ES(3) = '1' then println("Imm_ALU(0)");
                                                if ES(4) = '1' then println("Imm_ALU(1)");
                                                if ES(5) = '1' then println("Imm_ALU(2)");
                                                if ES(6) = '1' then println("Imm_ALU(3)");
                                                if ES(7) = '1' then println("Imm_ALU(4)");
                                                if ES(8) = '1' then println("Imm_ALU(5)");
                                                if ES(9) = '1' then println("Imm_ALU(6)");
                                        end;

if ES(23) AND ES(18) = '1' then  ** voter 9
                                        begin
                                                if ES(0) = '1' then println("Imm_ALU(7)");
                                                if ES(1) = '1' then println("Imm_ALU(8)");
                                                if ES(2) = '1' then println("Imm_ALU(9)");
                                                if ES(3) = '1' then println("Imm_ALU(10)");
                                                if ES(4) = '1' then println("Imm_ALU(11)");
                                                if ES(5) = '1' then println("Imm_ALU(12)");
                                                if ES(6) = '1' then println("Imm_ALU(13)");
                                                if ES(7) = '1' then println("Imm_ALU(14)");
                                                if ES(8) = '1' then println("Imm_ALU(15)");
                                                if ES(9) = '1' then println("Imm_ALU(16)");
                                        end;

if ES(23) AND ES(19) = '1' then  ** voter 10
                                        begin
                                                if ES(0) = '1' then println("Imm_ALU(17)");
                                                if ES(1) = '1' then println("Imm_ALU(18)");
                                                if ES(2) = '1' then println("Imm_ALU(19)");
                                                if ES(3) = '1' then println("Imm_ALU(20)");
                                                if ES(4) = '1' then println("Imm_ALU(21)");
                                                if ES(5) = '1' then println("Imm_ALU(22)");
                                                if ES(6) = '1' then println("Imm_ALU(23)");
                                                if ES(7) = '1' then println("Imm_ALU(24)");
                                                if ES(8) = '1' then println("Imm_ALU(25)");
                                                if ES(9) = '1' then println("Imm_ALU(26)");
                                        end;

if ES(23) AND ES(20) = '1' then  ** voter 11
                                        begin
                                                if ES(0) = '1' then println("Imm_ALU(27)");
                                                if ES(1) = '1' then println("Imm_ALU(28)");
                                                if ES(2) = '1' then println("Imm_ALU(29)");
                                                if ES(3) = '1' then println("Imm_ALU(30)");
                                                if ES(4) = '1' then println("Imm_ALU(31)");
                                                if ES(5) = '1' then println("Imm_ALU(0)");
                                                if ES(6) = '1' then println("Imm_ALU(1)");
                                                if ES(7) = '1' then println("Shft_Amnt(2)");
                                                if ES(8) = '1' then println("Shft_Amnt(3)");
                                                if ES(9) = '1' then println("Shft_Amnt(4)");
                                        end;

if ES(23) AND ES(20) = '1' then  ** voter 12
                                        begin
                                                if ES(0) = '1' then println("Dest_Reg(0)");
                                                if ES(1) = '1' then println("Dest_Reg(1)");
                                                if ES(2) = '1' then println("Dest_Reg(2)");
                                                if ES(3) = '1' then println("Dest_Reg(3)");
                                                if ES(4) = '1' then println("Dest_Reg(4)");
                                                if ES(5) = '1' then println("Dest_Reg(5)");
                                                if ES(6) = '1' then println("Dest_Reg(6)");
                                                if ES(7) = '1' then println("Dest_Reg(7)");
                                                if ES(8) = '1' then println("Dest_Reg(8)");
                                                if ES(9) = '1' then println("Dest_Reg(9)");
                                        end;

** Break down for IF Phase voters
if ES(22) AND ES(10) = '1' then ** voter 1
                                        begin
                                                if ES(0) = '1' then println("PC(0)");
                                                if ES(1) = '1' then println("PC(1)");
                                                if ES(2) = '1' then println("PC(2)");
                                                if ES(3) = '1' then println("PC(3)");
                                                if ES(4) = '1' then println("PC(4)");
                                                if ES(5) = '1' then println("PC(5)");
                                                if ES(6) = '1' then println("PC(6)");
                                                if ES(7) = '1' then println("PC(7)");
                                                if ES(8) = '1' then println("PC(8)");
                                                if ES(9) = '1' then println("PC(9)");
                                        end;

         if ES(22) AND ES(11) = '1' then ** voter 2
                                        begin
                                                if ES(0) = '1' then println("PC(10)");
                                                if ES(1) = '1' then println("PC(11)");
                                                if ES(2) = '1' then println("PC(12)");
                                                if ES(3) = '1' then println("PC(13)");
                                                if ES(4) = '1' then println("PC(14)");
                                                if ES(5) = '1' then println("PC(15)");
                                                if ES(6) = '1' then println("PC(16)");
                                                if ES(7) = '1' then println("PC(17)");
                                                if ES(8) = '1' then println("PC(18)");
                                                if ES(9) = '1' then println("PC(19)");
                                        end;

if ES(22) AND ES(12) = '1' then ** voter 3
                                        begin
                                                if ES(0) = '1' then println("PC(20)");
                                                if ES(1) = '1' then println("PC(21)");
                                                if ES(2) = '1' then println("PC(22)");
                                                if ES(3) = '1' then println("PC(23)");
                                                if ES(4) = '1' then println("PC(24)");
                                                if ES(5) = '1' then println("PC(25)");
                                                if ES(6) = '1' then println("PC(26)");
                                                if ES(7) = '1' then println("PC(27)");
                                                if ES(8) = '1' then println("PC(28)");
                                                if ES(9) = '1' then println("PC(29)");
                                        end;

if ES(22) AND ES(13) = '1' then ** voter 4
                                        begin
                                                if ES(0) = '1' then println("PC(30)");
```

```
                                    if ES(1) = '1' then println("PC(31)");
                                    if ES(2) = '1' then println("Inc_PC(0)");
                                    if ES(3) = '1' then println("Inc_PC(1)");
                                    if ES(4) = '1' then println("Inc_PC(2)");
                                    if ES(5) = '1' then println("Inc_PC(3)");
                                    if ES(6) = '1' then println("Inc_PC(4)");
                                    if ES(7) = '1' then println("Inc_PC(5)");
                                    if ES(8) = '1' then println("Inc_PC(6)");
                                    if ES(9) = '1' then println("Inc_PC(7)");
                              end;

if ES(22) AND ES(14) = '1' then ** voter 5
                              begin
                                    if ES(0) = '1' then println("Inc_PC(8)");
                                    if ES(1) = '1' then println("Inc_PC(9)");
                                    if ES(2) = '1' then println("Inc_PC(10)");
                                    if ES(3) = '1' then println("Inc_PC(11)");
                                    if ES(4) = '1' then println("Inc_PC(12)");
                                    if ES(5) = '1' then println("Inc_PC(13)");
                                    if ES(6) = '1' then println("Inc_PC(14)");
                                    if ES(7) = '1' then println("Inc_PC(15)");
                                    if ES(8) = '1' then println("Inc_PC(16)");
                                    if ES(9) = '1' then println("Inc_PC(17)");
                              end;

if ES(22) AND ES(15) = '1' then ** voter 6
                              begin
                                    if ES(0) = '1' then println("Inc_PC(18)");
                                    if ES(1) = '1' then println("Inc_PC(19)");
                                    if ES(2) = '1' then println("Inc_PC(20)");
                                    if ES(3) = '1' then println("Inc_PC(21)");
                                    if ES(4) = '1' then println("Inc_PC(22)");
                                    if ES(5) = '1' then println("Inc_PC(23)");
                                    if ES(6) = '1' then println("Inc_PC(24)");
                                    if ES(7) = '1' then println("Inc_PC(25)");
                                    if ES(8) = '1' then println("Inc_PC(26)");
                                    if ES(9) = '1' then println("Inc_PC(27)");
                              end;

if ES(22) AND ES(16) = '1' then ** voter 7
                              begin
                                    if ES(0) = '1' then println("Inc_PC(28)");
                                    if ES(1) = '1' then println("Inc_PC(29)");
                                    if ES(2) = '1' then println("Inc_PC(30)");
                                    if ES(3) = '1' then println("Inc_PC(31)");
                                    if ES(4) = '1' then println("Instr(0)");
                                    if ES(5) = '1' then println("Instr(1)");
                                    if ES(6) = '1' then println("Instr(2)");
                                    if ES(7) = '1' then println("Instr(3)");
                                    if ES(8) = '1' then println("Instr(4)");
                                    if ES(9) = '1' then println("Instr(5)");
                              end;

if ES(22) AND ES(17) = '1' then ** voter 8
                              begin
                                    if ES(0) = '1' then println("Instr(6)");
```

```
                                        if ES(1) = '1' then println("Instr(7)");
                                        if ES(2) = '1' then println("Instr(8)");
                                        if ES(3) = '1' then println("Instr(9)");
                                        if ES(4) = '1' then println("Instr(10)");
                                        if ES(5) = '1' then println("Instr(11)");
                                        if ES(6) = '1' then println("Instr(12)");
                                        if ES(7) = '1' then println("Instr(13)");
                                        if ES(8) = '1' then println("Instr(14)");
                                        if ES(9) = '1' then println("Instr(15)");
                              end;

if ES(22) AND ES(18) = '1' then ** voter 9
                              begin
                                        if ES(0) = '1' then println("Instr(16)");
                                        if ES(1) = '1' then println("Instr(17)");
                                        if ES(2) = '1' then println("Instr(18)");
                                        if ES(3) = '1' then println("Instr(19)");
                                        if ES(4) = '1' then println("Instr(20)");
                                        if ES(5) = '1' then println("Instr(21)");
                                        if ES(6) = '1' then println("Instr(22)");
                                        if ES(7) = '1' then println("Instr(23)");
                                        if ES(8) = '1' then println("Instr(24)");
                                        if ES(9) = '1' then println("Instr(25)");
                              end;

if ES(22) AND ES(19) = '1' then ** voter 10
                              begin
                                        if ES(0) = '1' then println("Instr(26)");
                                        if ES(1) = '1' then println("Instr(27)");
                                        if ES(2) = '1' then println("Instr(28)");
                                        if ES(3) = '1' then println("Instr(29)");
                                        if ES(4) = '1' then println("Instr(30)");
                                        if ES(5) = '1' then println("Instr(31)");
                              end;

print out all variables = ‘1’;
pop stack until empty;
end;
```

LIST OF REFERENCES

1. Lashomb, Peter A., "Triple Modular Redundant (TMR) Microprocessor System for Field Programmable gate Array (FPGA) Implementation," Master's Thesis, Naval Postgraduate School, Monterey, California, March 2002.

2. Ebert, Dean A., "Design and Development of a Configurable Fault Tolerant Processor (CFTP) for Space Applications," Master's Thesis, Naval Postgraduate School, Monterey, California, June 2003.

3. Johnson, Steven A., "Implementation of a Configurable Fault Tolerant Processor (CFTP)," Master's Thesis, Naval Postgraduate School, Monterey, California, March 2003.

4. Yuan, Rong, "Triple Modular Redundancy (TMR) in a Configurable Fault Tolerant Processor (CFTP) for Space Applications, Master's Thesis, Naval Postgraduate School, Monterey, California, December 2003.

5. Hennessy, John L. and Patterson, David A., *Computer Organization and Design, The Hardware/Software Interface*, Morgan Kaufmann, San Francisco, California, 1998.

6. Surratt, Mindy, "CFTP Development Environment Technical Manual", Naval Postgraduate School, Monterey California, December 2005.

7. Caffrey, Michael, Paul Graham, Eric Johnson, Michael Wirthlin, Nathan Rollins. Carl Carmichael, "Single-Event Upsets in SRAM FPGAs," Military and Aerospace Applications of Programmable Logic Devices, September 2002.

8. Wakerly, John F,. "Microcomputer Reliability Improvement Using Triple-Modular Redundancy", Proceedings of the IEEE, Vol. 64, No. 6, June 1976.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, ECE Department, Knorr
   Naval Postgraduate School
   Monterey, California

4. Professor Herschel H. Loomis
   Naval Postgraduate School
   Monterey, California

5. Professor Alan A. Ross
   Naval Postgraduate School
   Monterey, California

6. Ms. Mindy Surratt
   Naval Postgraduate School
   Monterey, California

7. Mr. Tim Meehan
   Naval Research Laboratory
   Washington, DC